UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series A

Practical methods for speeding-up the PNN method

Olli Virmajoki[1], Pasi Fränti[1]
and Timo Kaukoranta[2]

Report A-2000-2

ACM I.4.2, I.5.3
UDK 519.712
ISSN 0789-7316
ISBN 951-708-939-2

[1] *Department of Computer Science*
University of Joensuu
Box 111, FIN-80101 Joensuu
FINLAND

[2] *Turku Centre for Computer Science (TUCS)*
Department of Computer Science
University of Turku
FIN-20520 Turku, FINLAND

Practical methods for speeding-up the PNN method

(submitted to *Optical Engineering* 11.10.2000)

Olli Virmajoki[1], Pasi Fränti[1] and Timo Kaukoranta[2]

[1] *Department of Computer Science*
University of Joensuu
P.O. Box 111, FIN-80101 Joensuu
FINLAND

[2] *Turku Centre for Computer Science (TUCS)*
Dept. of Computer Science, University of Turku
Lemminkäisenkatu 14A, FIN-20520 Turku
FINLAND

Abstract: Pairwise nearest neighbor method (PNN) is a simple and well-known method for codebook generation in vector quantization. In its exact form, it provides good quality codebooks but at the cost of high run time. A fast exact algorithm has been recently introduced to implement the PNN in an order of magnitude faster than the original $O(N^3K)$ time algorithm. The run time, however, is still lower bounded by $O(N^2K)$, and therefore, additional speed-up may be required in applications where time is an important factor. We consider two practical methods to reduce the amount of work caused by the distance calculations. By experiments, we show that the run time can be reduced to 10-15% of original method for data sets in color quantization and in spatial vector quantization.

Keywords: Vector quantization, codebook generation, clustering algorithms, pairwise nearest neighbor method.

1. Introduction

Vector quantization (VQ) [1] is a method for reducing the amount of data. It can be applied in low bit rate compression of image and audio data, and in image analysis. The problem of generating a good codebook is one of the biggest problems in the design of a vector quantizer. The aim is to find a set of *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.

The most cited and widely used method for the codebook generation is the *Generalized Lloyd algorithm* (GLA) [2]. It starts with an initial codebook, which is iteratively improved until a local minimum is reached. The algorithm is easy to implement but it makes only local changes to the original codebook. The quality of the final codebook is therefore highly dependent on the initialization.

Better result can be achieved by the *pairwise nearest neighbor method* (PNN) [3]. It starts by initializing a codebook of size *N*, 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 codebook reduces to the desired size *M*. The PNN can also be combined with the GLA [4], used as a component in the *split-and-merge* algorithm [5], or as the crossover method in *genetic algorithm* [6].

The main drawback of the PNN is its slowness as the original implementation requires $O(N^3K)$ time [7]. An order of magnitude faster algorithm has recently been introduced [8], but the method is still lower bounded by $O(N^2K)$, which is more than the $O(NMK)$ time required by the GLA. Additional improvements are therefore needed in order to make the exact algorithm competitive also in speed.

Several speed-up methods have been introduced for the GLA [9-12]. The methods aim at reducing the computation required by the distance calculations. In PNN, the distance calculations are also the bottleneck of the algorithm. It is therefore expected that the ideas proposed for the GLA could also be adapted to the PNN. The difference is that the distance calculations in the PNN are not made in the Euclidean space. The distance of two particular clusters is defined as the increase of the distortion if the two clusters are merged. This difference is the primary reason for the possible success or failure of the speed-up methods in the PNN.

We consider two different speed-up methods. The *partial distortion search* (PDS) by Bei and Gray [10] terminates a single distance calculation immediately when the partially distance exceeds the shortest distance found so far. This idea is independent on the chosen metrics and therefore it can be directly applied to the PNN, too.

The second method is the *mean-distance-ordered partial search* (MPS) technique introduced by Ra and Kim [12]. It uses the component means of the vectors for deriving a pre-condition for the distance calculations and, in this way, a large number of the distance calculations can be omitted completely. We will show that similar pre-condition can be derived for the distance calculations in the PNN.

The rest of the paper is organized as follows. The PNN method and its fast exact implementation is given in Section 2. We give a detailed description of the method, in order to allow the reader to implement the exact PNN accurately. New speed-up methods are then introduced in Section 3. In particular, we introduce the PDS and the MPS methods in the context of the PNN. Simulation results for various training sets are shown in Section 4. Experiments show that the run time can be reduced to 10-15% in the case of the two favorable data sets, whereas only moderate improvement can be achieved in the case of the unfavorable data set. Conclusions are drawn in Section 5.

2. Pairwise nearest neighbor method

We consider a set of N training vectors (x_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 Euclidean distance between the training vectors and their representative code vectors:

$$f(C) = \frac{1}{N}\sum_{i=1}^{N}\left\|x_i, c_{p_i}\right\|^2 \tag{1}$$

where p_i is the cluster (partition) index of the training vector x_i. *Cluster* is defined as the set of training vectors that belong to the same partition a:

$$s_a = \left\{ x_i \middle| p_i = a \right\} \tag{2}$$

The basic structure of the PNN is shown in Fig. 1. The method starts by initializing each training vector x_i as its own code vector c_i. In each step of the algorithm, the size of the codebook is reduced by merging two clusters. The cost of merging two clusters s_a and s_b can be calculated as [3]:

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

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.

The exact variant of the PNN applies local optimization strategy: all possible cluster pairs are considered and the one (a, b) increasing the distortion least is chosen:

$$a,b = \arg \min_{\substack{i,j \in [1,N] \\ i \neq j}} d_{i,j} \tag{4}$$

The clusters are then merged and the process is repeated until the codebook reaches the size M. Straightforward implementation of this takes $O(N^3K)$ time because there are $O(N)$ steps, and in each step there are $O(N^2)$ cost function values to be calculated.

```
PNN(X, M) →C, P
    s_i ←{x_i} ∀i∈[1,N];
    m ←N;
    REPEAT
        (s_a, s_b) ←NearestClusters();
        MergeClusters(s_a, s_b);
        m ←m-1;
        UpdateDataStructures();
    UNTIL m=M;
```

Fig. 1. Structure of the exact PNN method.

2.1. Fast exact PNN

A much faster variant of the PNN can be implemented by maintaining for each cluster a pointer to its nearest neighbor [8]. The nearest neighbor nn_a for a cluster s_a is defined as the cluster minimizing the merge cost:

$$nn_a = \arg \min_{\substack{j \in [1,N] \\ j \neq a}} d_{a,j} \tag{5}$$

In this way, only few nearest neighbor searches are needed in each iteration. The method is denoted as *fast exact PNN*, and its implementation details are given next.

For each cluster s_j, we also maintain the cluster size n_j, the corresponding code vector c_j, and the pointer to its nearest neighbor nn_j. The nearest neighbor pointer is assigned with the cost value d_j indicating the amount of increase in distortion if the cluster s_j is merged to s_{nnj}. For each training vector, we maintain the index of the cluster p_i, which it belongs to.

2.2. Initialization

In the initialization, each training vector x_i is assigned to its own cluster. The corresponding cluster size is set to 1, and the code vector c_i as the training vector itself:

$$p_i \leftarrow i \;\; ; \;\; n_i \leftarrow 1 \;\; ; \;\; c_i \leftarrow x_i \;\; \forall\, i \in [1, N] \tag{6}$$

In order to generate the nearest neighbor table, we must find the nearest neighbor nn_i for every cluster. This is done by considering all other clusters as tentative neighbor and selecting the one that minimizes (3). There are $O(N^2)$ pairs to be considered, and thus, the initialization phase takes $O(N^2K)$ time in total.

2.3. Merging the clusters

The optimal cluster pair (s_a and s_b) to be merged is the two clusters having the minimum d_j-value and its nearest neighbor nn_j:

$$a \leftarrow \arg\min_{j \in [1,N]} d_j \;\; ; \;\; b \leftarrow nn_a \tag{7}$$

This pair can be found in $O(N)$ time using linear search for the nearest neighbor table. The merge of the clusters is then performed as follows. First, we update the partition indices so that the combined cluster replaces s_a, and the cluster s_b becomes obsolete:

$$p_i \leftarrow a \quad \forall\, i \,\big|\, p_i = b \tag{8}$$

The size of the merged cluster is calculated as:

$$n_a \leftarrow \; n_a + n_b \tag{9}$$

The code vector of the combined cluster could be calculated as the weighted average of c_a and c_b:

$$c_a \leftarrow \; \frac{n_a c_a + n_b c_b}{n_a + n_b} \tag{10}$$

However, as we also maintain the partition index of each training vector p_i, it is therefore better to calculate the new code vector as the centroid of the cluster in order to minimize rounding errors:

$$c_a \leftarrow \; \frac{1}{n_a} \cdot \sum_{p_i = a} x_i \tag{11}$$

These steps can be performed at most in $O(NK)$ time.

2.4. Updating the nearest neighbor pointers

The nearest neighbor nn_a for the merged cluster (now s_a) must be resolved by calculating the distance function values (3) between the new cluster and all other remaining clusters. This can be performed in $O(NK)$ time.

The nearest neighbor function is not symmetrical, i.e. $nn_a=b$ does not imply $nn_b=a$. Therefore, we must also resolve the nearest pointer for all clusters whose nearest neighbor before the merge was either a or b ($nn_i=a$ or $nn_i=b$). This takes $O(NK)$ time for a single cluster and there are approximately 3-5 clusters on average to be updated in each step of the algorithm, according to [8]. The overall time complexity of the update is therefore $O(\tau NK)$, where τ denotes the number of clusters whose the nearest neighbor pointer must be resolved. To sum up, the time complexity of the fast exact PNN is $O(\tau N^2 K)$.

2.5. Lazy PNN

The number of distance calculations can be reduced by delaying the update of the nearest neighbor pointers. This idea is based on the *monotony property* shown in [13], which says that the minimum cluster distances never decrease due to the merge of the optimal pair. For example, assume that the nearest neighbor for a cluster s_i was s_a before the merge, and s_c after the merge. From the monotony property we know that $d_{i,a} \leq d_{i,c}$. We therefore do not need the exact distance but the previous distance serves as a lower bound. In practice, we can assume that the nearest neighbor after the merge is s_{a+b}, and use the previous cost function value $d_{i,a}$ (or $d_{i,b}$). The distance value is labeled as "outdated", and it will be updated only if it becomes the candidate of being the smallest distance of all. In this way, we can reduce the computation by about 35% while preserving the exactness of the PNN [13].

3. Speed-up methods for the PNN

There are two alternative approaches for speeding-up the PNN. One approach is to sacrifice the exactness of the PNN either by using a faster but sub optimal method for selecting the clusters to be merged [3], or by generating an initial codebook of size $N>M_0>M$ before the PNN [4]. However, we take another approach, in which the exactness of the PNN is preserved in all steps of the algorithm.

The main loop of the PNN reduces the number of code vectors from N to M. It seems to be impossible to reduce the number of stages in this loop without sacrificing the optimality of the steps. We therefore aim at reducing the computation inside the loop. The loop consists of the search of the cluster pair, the merge, and the update of the nearest neighbor pointers. The search takes $O(N)$, and the merge $O(NK)$ time. In the update phase, we find nearest neighbor for τ clusters, and every search requires $O(N)$ distance calculations. Thus, the update phase requires $O(\tau NK)$ in total, and it is clearly the bottleneck of the algorithm.

We consider the following two methods

- *partial distortion search* (*PDS*) [10], and
- *mean-distance-ordered partial search* (*MPS*) [12].

The methods are tailored for gaining speed without sacrificing the optimality. They are new in the context of the PNN but widely used in the search of nearest code vector in the GLA, and in the encoding phase of VQ. In the PNN, they apply both in the initialization stage, and in the main loop of the PNN. The methods can be considered practical as they achieve speed-up without complicated data structures, without excessive increase of the memory-consumption, and they are easy to implement.

3.1. Partial distortion search

Let s_a be the cluster, for which we seek the nearest neighbor. We use full search, i.e., calculate the distance values $d_{a,j}$ between s_a and all other clusters s_j. Let d_{min} be the distance of the best candidate found so far. The distance is calculated cumulatively by summing up the squared differences in each dimension. In partial distortion search, we utilize the fact that the cumulative summation is non-decreasing, as the individual terms are non-negative. The calculation is therefore terminated and the candidate rejected if the partial distance value exceeds the current minimum d_{min}.

The implementation of the partial distance calculation is shown in Fig. 2. The distance function consists of the *squared Euclidean distance* ($e_{a,j}$) of the cluster centroids, and a weighting factor ($w_{a,j}$) that depends on the cluster sizes:

$$e_{a,j} = \sum_{k=1}^{K} \left(c_{ak} - c_{jk} \right)^2 \tag{12}$$

$$w_{a,j} = \frac{n_a \cdot n_j}{n_a + n_j} \tag{13}$$

Here c_{ak} and c_{jk} refer to the k'th component of the corresponding vector. The weighting factor $w_{a,j}$ is calculated first, and the squared Euclidean distance $e_{a,j}$ is then cumulated by summing up the squared differences in each dimension. After each summation, we calculate the partial distortion value ($w_{a,j} \cdot e_{a,j}$) and compare it to the distance of the best candidate (d_{min}):

$$w_{a,j} \cdot e_{a,j} \geq d_{min} \tag{14}$$

The distance calculation is terminated if this condition is found to be true. The calculations of the partial distortion require an additional multiplication operation and an extra comparison for checking the termination condition. We refer this as the *simple variant*. Speed-up can be achieved if this extra work does not exceed the time saved by the termination. The extra multiplication can be avoided by formulating the termination condition as:

$$e_{a,j} \geq \frac{d_{min}}{w_{a,j}} \tag{15}$$

The right part of the equation can now be calculated in the beginning of the function, and only the comparison remains inside the summation loop. We refer this as the *optimized variant*. As a drawback, there are one extra division due to (15) and extra multiplication outside the loop.

The computational efficiency of the two variants are compared to that of the full search in Table 1. The simple variant is faster when the dimensions are very small and in cases when the termination happens earlier. The equation (14) is also less vulnerable to rounding errors than (15). The optimized variant, on the other hand, produce significant improvement when the dimensions are very large.

Overall, the effectiveness of the method depends on the quality of the current candidate. It is therefore important to have a good initial candidate so that the $d_{\min}$ would be small already at the early stages of the calculations. In this way, more distance calculations can be terminated sooner. In the previous iteration, the nearest neighbor for s_a was one of the clusters that were merged. It is expected that the distance to the merged cluster remains relatively small and, therefore, we take this as the first candidate. This minor enhancement turns out to provide significant improvement in the algorithm, see Section 4.

```
MergeCost(s_a, s_j, d_min) →d;
    e ← 0;
    k ← 0;
    w ← n_a · n_j / (n_a + n_j);
    REPEAT
        k ← k + 1;
        e ← e + (c_ak - c_jk)^2;
        d ← w · e;
    UNTIL (d > d_min) OR (k = K);
    RETURN d;
```

Fig. 2. Pseudo code for the distance calculation in the (simple) PDS method.

Table 1: Summary of the arithmetic operations involved in the distance calculations. The value $q \in [0,1]$ refers as the proportion of the processed dimensions.

Variant:	*	/	+
Full search	$k + 2$	1	$2k + 1$
Simple variant	$2kq + 1$	1	$2kq + 1$
Optimized variant	$kq + 2$	2	$2kq + 1$

3.2. Mean-distance ordered partial search

The mean-distance-ordered partial search [12] applies two different techniques to speed-up the search of the nearest code vector. Firstly, it uses a fast pre-condition for checking whether the distance calculation to a given candidate cluster can be omitted. Secondly, it sorts the codebook according to the component means of the code vectors and derives limits for the search.

The method stores the component sums of each code vector. Let s_a be the cluster, for which we seek its nearest neighbor, and s_j the candidate cluster to be considered. The

distance of their corresponding code vectors c_a and c_j can be approximated by the squared distance of their component sums:

$$\hat{e}_{a,j} = \left(\sum_{k=1}^{K} c_{ak} - \sum_{k=1}^{K} c_{jk} \right)^2 \tag{16}$$

The component sums correspond the projections of the code vectors to the diagonal axis of the vector space. In typical training sets, the code vectors are highly concentrated along the diagonal axis, and therefore, the distance of their component sums highly correlate to their real distance. The following inequality holds true [12]:

$$\hat{e}_{a,j} \leq K \cdot e_{a,j} \tag{17}$$

This inequality has been utilized in the search of nearest neighbor in VQ by deriving the following pre-condition:

$$K \cdot e_{\min} < \hat{e}_{a,j} \tag{18}$$

In other words, if the squared Euclidean distance of the component sums exceeds the distance to the best candidate found so far (multiplied by K), the real distance cannot be smaller than $e_{\min}$, according to (17). This is illustrated in Fig. 3, where the distance from A to B is the current minimum. All potential candidates and their projections must therefore lie inside the circle. The pre-condition can be calculated fast in an $O(1)$ time as the component sums are already known. If the pre-condition (18) is true, the candidate code vector can be rejected without the distance calculation.

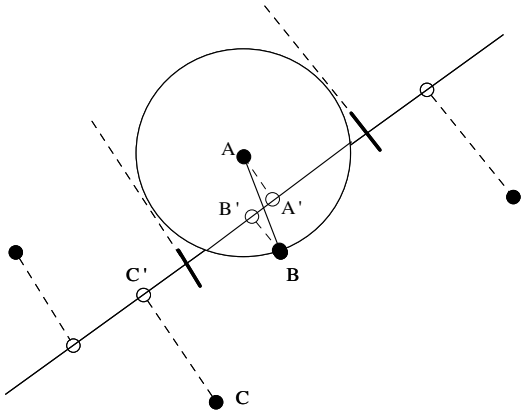

Fig. 3. Code vectors (black dots) and their projections (empty dots) according to the component sums.

In the PNN, the distance function consists of the squared Euclidean distance ($e_{a,j}$) of the code vectors and the weighting factor ($w_{a,j}$). As shown in (14), these two can be calculated separately. The inequality (17) can therefore be generalized to the cluster distances as:

$$w_{a,j} \cdot \hat{e}_{a,j} \leq K \cdot w_{a,j} \cdot e_{a,j} \tag{19}$$

Given the minimum distance $e_{\min}$, we can then derive similar pre-condition for the PNN distance function as:

$$K \cdot d_{\min} < w_{a,j} \cdot \hat{e}_{a,j} \tag{20}$$

This can be applied as follows. The clusters are processed in any given order. The weighting factor $w_{a,j}$ and the distance of the component sums ($\hat{e}_{a,j}$) are first calculated, and the pre-condition is evaluated. If it holds true, the calculation of the Euclidean distance can be omitted and the candidate cluster s_j rejected.

Further speed-up can be obtained by sorting the codebook according to their component sums, and then proceed the clusters in the order given by the sorting [12]. The search starts from the cluster s_a and it proceeds bidirectionally along the projection axis. When we find the first cluster for which the pre-condition is met, we know that all the rest code vectors in that particular direction will meet the pre-condition of (18). In the GLA, this gives definite boundaries for the search and the rest of the candidates can be rejected even without calculation of the pre-condition.

In the PNN, we cannot make solid bounds for the search because of the weighting factor (13) involved in the distance function. Consider the situation in Fig. 4. The clusters B and C (with cluster sizes 2 and 3) are the first two that meet the inequality (20). However, they do not necessarily bound the search as there can be smaller clusters further away, for which the distance to A is smaller. In the example of Fig. 4, there is one such cluster; the one between C and E.

The pre-condition cannot be used as such for giving bounds for the search but we have found a weaker condition to terminate the search. The pre-condition guarantees that there are no more potential clusters in the respective direction, whose size is greater than or equal to the cluster, who met the pre-condition. In specific, if the size of the candidate cluster equals to one and the inequality (20) holds, we can terminate the search in that direction. For example, in Fig. 4 the search bounds will be the clusters D and E because they are the first clusters of size 1 in the corresponding directions, and the inequality (20) holds for them.

The pseudo code of the algorithm is given in Fig. 5. For simplicity, we assume that the clusters have already been sorted before the call of the routine.

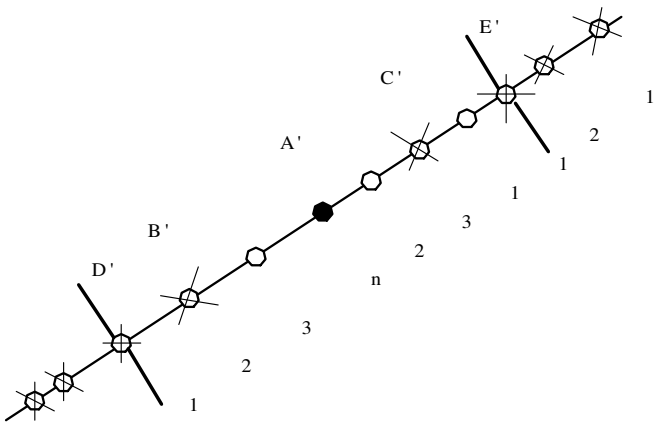

Fig. 4. An example of setting up the search boundaries in the MPS. The projections of the clusters are drawn as circles. The cluster A (with black dot) is the one for which we search the nearest neighbor. The numbers represent the cluster sizes, and the crossing indicates that the pre-condition hold true for the cluster and its distance calculation can be omitted.

```
PreCondition(s_a, s_j, d_min) →BOOLEAN;
    w ← n_a · n_i / (n_a + n_i);
    ê ← (sum_a - sum_i)^2;
    RETURN( K · d_min < w · ê );

CheckCandidate(s_a, s_i, n_a, d_min, nn, direction);
  IF PreCondition(s_a, s_i, d_min) THEN
      IF n_a = 1 THEN direction ← FALSE
  ELSE
      d ←MergeCost(s_a, s_j, d_min);
      IF d < d_min THEN
          d_min ←d;
          nn ←j;
  RETURN;

SearchNearestMPS(c_a, c_j, d_min) →nn_a, d_a;
    d_min ← ∞;
    up ← TRUE;
    down ← TRUE;
    j_1 ← a;
    j_2 ← a;

    WHILE (up OR down) DO

      IF up THEN
          j_1 ← j_1 + 1;
          IF j_1 > N THEN up ← FALSE
          ELSE CheckCandidate(s_a, s_i1, n_a, d_min, nn, up);

      IF down THEN
          j_2 ← j_2 - 1;
          IF j_2 < 1 THEN down ← FALSE
          ELSE CheckCandidate(s_a, s_i2, n_a, d_min, nn, down);

      END-WHILE;

    RETURN nn, d_min;
```

Fig. 5. Pseudo code for the MPS method used in the PNN.

3.3. Initialization with the methods

When special speed-up techniques are not used, the initial nearest neighbors nn_i and the associated distances d_i can be determined by calculating only half of the pairwise distances $d_{i,j}$ because the merge cost function (3) is symmetrical. When we are determining the nearest neighbor for cluster s_a and we have calculated its distance to cluster s_b, we check also if s_a closer to s_b than s_b's current nearest neighbor nn_b, i.e. $d_{a,b}<d_b$. If that is the case we update the nearest neighbor of cluster s_b also.

However, this can not be done if the PDS is utilized in the initialization. Because then the real distance between clusters s_a and s_a is not always calculated but only the partial distance, which can be smaller than the real one. Therefore it can not be used for the update of the nearest neighbor of cluster s_b.

Fortunately, it is not necessary that all nearest neighbor pointers have been assigned to really nearest cluster to preserve the exactness of the PNN. It is enough that for each

cluster s_i, we have determined its nearest neighbor among the clusters whose index j is greater than i. Thus, the nearest neighbor for cluster s_i is

$$nn_i = \arg\min_{i<j\le N} d_{i,j}. \tag{21}$$

This guarantees that nearest neighbor pair of all cluster pairs is stored. The second nearest cluster pair is not necessarily stored but it will not be needed anyway. After the merge of the nearest cluster pair, we update the nearest neighbor pointers for the appropriate clusters by considering all clusters. Therefore, we always have the knowledge on the nearest cluster pair.

4. Test results

We generated training sets from six images: *Bridge*, *Camera*, *Miss America*, *Table tennis*, *House* and *Airplane*, see Fig. 6. The vectors in the first two sets (*Bridge*, *Camera*) are 4×4 pixel blocks from the gray-scale images. The third and fourth sets (*Miss America*, *Table Tennis*) 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 (*House*, *Airplane*) consist of color values of the *RGB* images. Applications of this kind of data sets is found in image and video image coding (*Bridge*, *Camera*, *Miss America*, *Table tennis*), and in color image quantization (*House*, *Airplane*).

Spatial vectors:		**Spatial residual vectors:**		**Color vectors:**	
Bridge (256×256) *K*=16, *N*=4096	*Camera* (256×256) *K*=16, *N*=4096	*Miss America* (360×288) *K*=16, *N*=6480	*Table tennis* (720×486) *K*=16, *N*=21771*	*Airplane* (512×512) *K*=3, *N*=77274*	*House* (256×256) *K*=3, *N*=34112*

Fig. 6. Sources of the training sets. *Duplicate training vectors are combined and frequency information is stored.

4.1. Experiments with the PDS variants

The effect of the initialization of the PDS is first studied as a function of the vector dimension. For this purpose, we use artificially generated training sets with the parameters N=1024, M=256, and varying the vector size K from 16 to 1024. The results are shown in Fig. 7, and they clearly demonstrate the importance of the initial guess for training sets with large vector dimension. The improvement is less significant for training sets with K<16 but still large enough to be useful. In the following, we assume that the initial guess is always used.

The performance of the two PDS variants (simple and the optimized variant) are

summarized in Fig. 8 for the six training sets of Fig. 6. The results shows that the simple variant is better for the training sets (*House* and *Airplane*) with small vector dimensions (K=3). This is because of the extra division operation in the optimized variant. In fact, the optimized variant is even slower than the original PNN. For the other training sets (*Bridge*, *Camera*, *Miss America*, *Table tennis*), the optimized variants works much better and gives always equal to or better result than the simple variant. The significance of the division operation is much smaller in case of vector with higher dimension.

We would also like to note, that the cost of the division operation is hardware dependent. The results indicate that the relative performance of the optimized variant increases as a function of K, but one should not draw any final conclusions about the exact value of K, after which the optimized variant outperforms the simple variant.

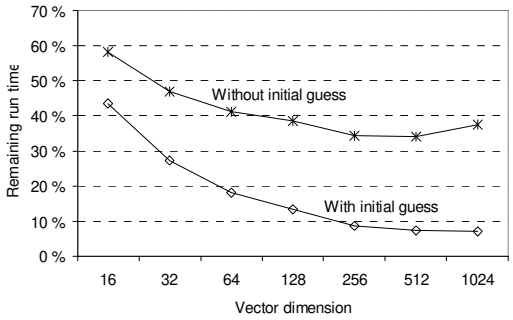

Fig. 7. Remaining run time relative to full search PNN for the optimized PDS with and without initial guess.

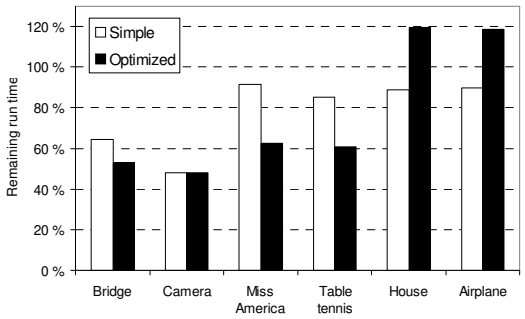

Fig. 8. Remaining run time relative to full search PNN for the simple and optimized variants of the PDS.

4.2. Experiments with the MPS variants

The effectiveness of the MPS is shown in Table 2 as the number of operations relative to the full search. The first observation is that the use of the pre-condition is efficient for *Bridge* and *Camera* (run time is decreased down to about 30%) but only moderate improvement is achieved with the other four training sets (down to about 70-80%). The second observation is that the sorting gives significant improvement for *Bridge*, *Camera*, *Airplane* and *House*. For the color images (*Airplane* and *House*), the sorting is important as it eliminates most of the pre-condition tests; only about 2% of the tests remains.

Overall, the MPS works very well except for the residual training sets (*Miss America*, *Table tennis*). For these, the MPS can reduce the computation only down to about 2/3 from that of the full search. This originates from the nature of the training sets; the training vectors are relatively evenly distributed in the vector space and there are no clear clusters in the training sets. The projection of the vectors onto the diagonal axis therefore is not as efficient as it would be if the vectors were clustered in the vector space.

Table 2. The remaining (relative) run times for the MPS variants.

Relative	MPS without sorting			MPS with sorting		
	Pre-conditions /search	Distance calculations /search	Remaining run time (%)	Pre-conditions /search	Distance calculations /search	Remaining run time (%)
Bridge	100 %	17.5 %	32.9 %	16.6 %	15.3 %	16.5 %
Camera	100 %	9.6 %	28.8 %	8.5 %	7.9 %	11.0 %
Miss America	100 %	82.1 %	72.1 %	80.5 %	78.6 %	66.4 %
Table tennis	100 %	85.3 %	71.4 %	84.4 %	83.9 %	71.3 %
Airplane	100 %	4.9 %	77.2 %	1.7 %	1.2 %	18.6 %
House	100 %	6.9 %	78.3 %	2.4 %	1.8 %	15.2 %

Absolute	MPS without sorting			MPS with sorting		
	Pre-conditions /search	Distance calculations /search	Remaining run time (%)	Pre-conditions /search	Distance calculations /search	Remaining run time (%)
Bridge	2599.1	455.2	32.9	432.6	397.4	16.5
Camera	2575.1	247.6	28.8	218.2	202.9	11.0
Miss America	3959.2	3252.1	72.1	3187.0	3112.1	66.4
Table tennis	13021.4	11103.4	71.4	10983.7	10922.2	71.3
Airplane	48189.5	2373.6	77.2	805.4	560.4	18.6
House	21285.1	1464.9	78.3	506.9	373.2	15.2

4.3. Joint experiments

Table 3 gives the detailed work load of the different PNN variants for three training sets, one from each category. The results show that the joint use of the PDS and MPS is successful. Only in the case of the first category (*Bridge* and *Camera*), the use of the MPS somewhat weakens the effect of the PDS. This is because with the MPS, most of the distance calculations are done to nearby clusters and therefore, the PDS is less efficient. Nevertheless, the overall improvement is still very good for these training sets, too. For the other four training sets, the use of the MPS do not decrease the effectiveness of the PDS.

The overall run times of the PDS and MPS are summarized in Table 4. The methods are also combined with the *Lazy PNN* of Section 2.5. The joint use of these three methods reduces the run time down to 10-15% in the case of the four favorable training sets (*Bridge*, *Camera*, *House*, *Airplane*), but only down to about 70% in the case of residual vectors (*Miss America*, *Table tennis*). The use of the lazy update (*Lazy PNN*) gives further reduction of about 15-30%.

To sum up, if we combine all the idea presented in this paper (PDS, MPS, Lazy update, initialization trick of Section 3.3), we can reduce the run time down to 8-15% in the case of the favorable training sets (*Bridge*, *Camera*, *House*, *Airplane*), and down to about 50% in the case of the unfavorable sets (*Miss America*, *Table tennis*).

From the experiments we can see that the results greatly depend on the training set. An important parameter of the training set is the size (dimension) of the vectors. The effect of the vector size is therefore demonstrated in Fig. 9 with the artificial training sets described earlier in this section. Obviously, the overall run time increases as a function of the vector size. On the other, the relative improvement of the speed-up methods also increases, which partly compensates the increase in the vector size.

Table 3. The average number of distance calculations per nearest neighbor search, the average number of processed vector dimension during the distance calculation, and the total number of processed vector dimensions per search on average.

Bridge, N=4096, M=256, K=16			
	Distance Calculations / search	Dimensions / distance calculation	Dimensions / search
Full	2208.6	16.0	35338.3
PDS	2208.7	3.0	6534.2
MPS + PDS	397.4	5.7	2261.0

House, N=34112, M=256, K=3			
	Distance Calculations / search	Dimensions / distance calculation	Dimensions / search
Full	16514.8	3.0	49544.5
PDS	16533.6	1.1	17538.3
MPS + PDS	373.2	1.2	434.7

Miss America, N=6480, M=256, K=16			
	Distance Calculations / search	Dimensions / distance calculation	Dimensions / search
Full	3404.8	16.0	54476.9
PDS	3401.5	4.8	16194.3
MPS + PDS	3112.1	4.8	14805.4

Table 4. Run times (in seconds) for the six training sets (M=256).

	Bridge		House		Miss America	
	PNN	Lazy-PNN	PNN	Lazy-PNN	PNN	Lazy-PNN
Full	79	52	1524	1126	229	145
PDS	42	28	1826	1318	143	91
MPS + PDS	13	10	231	195	152	108
	Camera		Airplane		Table tennis	
	PNN	Lazy-PNN	PNN	Lazy-PNN	PNN	Lazy-PNN
Full	73	51	8812	6237	2895	1816
PDS	35	25	10460	7145	1756	1109
MPS + PDS	8	6	1636	1295	2063	1462

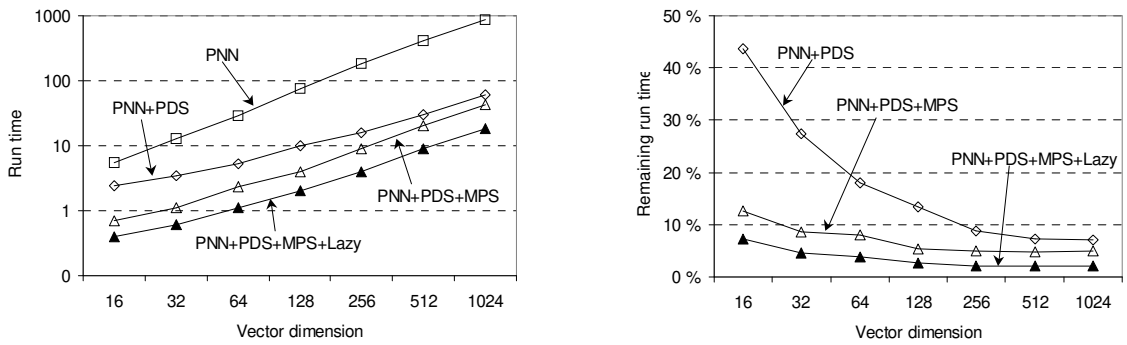

Fig. 9. Run time (in seconds) as a function of the code vector dimension (left), and remaining run time relative to the full search PNN (right).

4.4. Integration with other algorithms

The improved PNN benefits also other hybrid codebook generation methods. Here we have tested two of them: (i) GLA-PNN-GLA and (ii) GA-PNN. The first method is a hybridization of the PNN and GLA due to [4]. It starts with an initial codebook of size $M_0 > M$, which is generated by the GLA. The resulting codebook is then reduced to the final size using the exact PNN, and finally fine-tuned by the GLA, again. The method is parametrized by the choice of M_0, and is denoted here as the *GLA-PNN-GLA*. The GLA is implemented as in [9].

The second method is the *genetic algorithm* (GA) as proposed in [6]. The PNN is used in the implementation of the crossover as described in [14]. The method is based on evolutionary computing; it uses crossover for creating new candidate solutions, and selection for directing the search towards better quality codebooks. The role of the PNN is to provide high quality candidate codebooks instead of using exhaustive trial-and-error approach. We denote this method as the *GA-PNN*.

The performance of these two methods are compared in Fig. 10 using *Bridge*. The results clearly demonstrate that better quality codebooks can be obtained by the hybrid methods than using the PNN alone. The choice between these two methods depends on whether the user prefers speed or quality. The GA-PNN is capable of providing the best codebooks at the cost of higher run time whereas the GLA-PNN-GLA is the better choice if the time is limited.

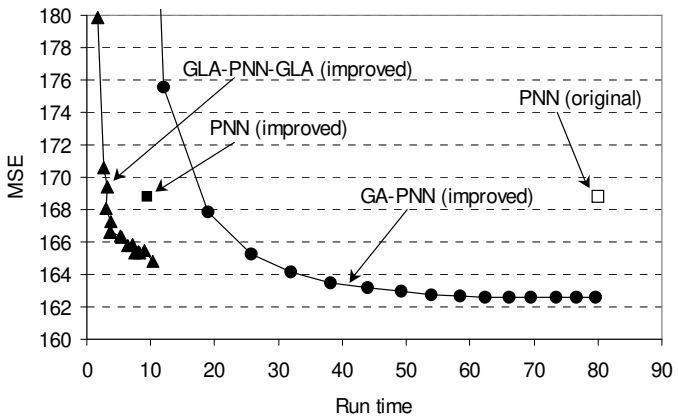

Fig. 10. Time-distortion performance of the hybrid algorithms for *Bridge*. The GLA-PNN-GLA is parametrized by changing the size of the initial codebook from 256 to 4096. The results of the GA-PNN are shown from the first 15 iterations.

5. Conclusions

We have considered two different speed-up methods for the PNN: the partial distortion search (PDS), and the mean-distance-ordered partial search (MPS). The methods can be considered practical as they achieve the speed-up without complicated data structures, without excessive increase of the memory-consumption, and they are easy to implement.

The PDS method works well in most cases and achieves similar speed-up than was obtained within the GLA; roughly about 50% with the favorable training sets. The use of the PDS is questionable only with vectors of very small dimension. In this case, the improvement can be overwhelmed by the overhead caused by the additional test. The use of the initial guess was also found to be important. Overall, the PDS is very efficient with vectors of very large dimension. For example, less than 10% run time is required with vectors of size 256 or more.

The MPS works also very well in most cases by reducing the run time down to 10-20%. The exception is the set of residual vectors, for which only moderate improvement was obtained. The improvement is not as good as within the GLA for two reasons: (i) the calculation of the pre-condition is more complicated and takes more time, and (ii) the pre-condition cannot be used for giving absolute bounds for the search as in the GLA.

To sum up, if we combine all the speed-up methods discussed in this paper, we can reduce the run time down to 8-15% in the case of the four favorable training sets, and down to about 50% in the case of the unfavorable sets (residual vectors). In the case of vectors with very large dimension (256 or greater) the run time of the favorable sets can be reduced down to 2%.

We also demonstrated that the improvements are applicable within more sophisticated hybrid methods, in which the PNN is used as a component. Two such methods were considered: the GLA-PNN-GLA, and the genetic algorithm with PNN crossover.

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. W.H. Equitz, "A new vector quantization clustering algorithm", *IEEE Transactions on Acoustics, Speech, and Signal Processing* **37** (10), 1568-1575, October 1989.
4. 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.
5. T. Kaukoranta, P. Fränti and O. Nevalainen, "Iterative split-and-merge algorithm for VQ codebook generation", *Optical Engineering*. **37** (10), 2726-2732,October 1998.
6. P. Fränti, J. Kivijärvi, T. Kaukoranta and O. Nevalainen, "Genetic algorithms for large scale clustering problem", *The Computer Journal*, **40** (9), 547-554, 1997.
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. P. Fränti, T. Kaukoranta, D.-F. Shen and K.-S. Chang, "Fast and memory efficient implementation of the exact PNN", *IEEE Transactions on Image Processing*, **9** (5), 773-777, May 2000.
9. T. Kaukoranta, P. Fränti and O. Nevalainen, "A fast exact GLA based on code vector activity detection", *IEEE Trans. on Image Processing*, **9** (8), 1337-1342, August 2000.
10. C.-D. Bei and R.M. Gray, "An improvement of the minimum distortion encoding algorithm for vector quantization", *IEEE Transactions on Communications*, **33** (10), 1132-1133, October 1985.
11. S.-H. Chen and W.M. Hsieh, "Fast algorithm for VQ codebook design", *IEE Proceedings-I*, **138** (5), 357-362, October 1991.
12. S.-W. Ra and J.-K. Kim, "A Fast Mean-Distance-Ordered Partial Codebook Search Algorithm for Image Vector Quantization", *IEEE Transactions on Circuits and Systems*, **40** (9), 576-579, September 1993.
13. T. Kaukoranta, P. Fränti and O. Nevalainen, "Vector quantization by lazy pairwise nearest neighbor method", *Optical Engineering*, **38** (11), 1862-1868, November 1999.
14. P. Fränti, "Genetic algorithm with deterministic crossover for vector quantization", *Pattern Recognition Letters*, **21** (1), 61-68, January 2000.

Appendix: C-sources of the PDS-routines

```
/*====================================================================*/

static llong MergeDistortion(CODEBOOK* CB, int p1, int p2)
{
  /* Original inline */
  int i=0, Vsize;
  llong   dist=0LL, diff, dist_limit=MAXLLONG;
  double  factor;

  Vsize = VectorSize(CB);
  factor = ((double)VectorFreq(CB,p1) * (double)VectorFreq(CB,p2)) /
            ((double)VectorFreq(CB,p1) + (double)VectorFreq(CB,p2))
       /* Precondition: Vsize >= 1 */
      /* if dist >= dist_limit, returns at least factor * dist_limit */
  do
       {
          diff = (llong)VectorScalar(CB,p1,i) -
                (llong)VectorScalar(CB,p2,i);
          dist += diff * diff;
       } while(dist < dist_limit && ++i < Vsize);
  dist = round( factor * (double)dist);

  return( dist);
}


/*--------------------------------------------------------------------*/

static llong MergeDistortionPds(CODEBOOK* CB, int p1, int p2,
                                        llong dist_limit)
{
  /* Variant simple */
  int i=0, Vsize;
  llong   dist=0LL, diff, distortion=0LL;
  double  factor;

  Vsize = VectorSize(CB);
  factor = ((double)VectorFreq(CB,p1) * (double)VectorFreq(CB,p2)) /
           ((double)VectorFreq(CB,p1) + (double)VectorFreq(CB,p2));
  /* Precondition: Vsize >= 1 */
  /* if distortion >= dist_limit, returns at least dist_limit */
  do    {
          diff = (llong)VectorScalar(CB,p1,i) -
                (llong)VectorScalar(CB,p2,i);
          dist += diff * diff;
          distortion = round( factor * (double)dist);
       } while( distortion < dist_limit && ++i < Vsize);

  return( distortion);
}


/*--------------------------------------------------------------------*/

static llong MergeDistortionPds(CODEBOOK* CB, int p1, int p2,
                                        llong maxdist)
{
  /* Variant Optimized */
  int i=0, Vsize;
  llong   dist=0LL, diff, distortion=0LL, dist_limit;
  double  factor;

  Vsize = VectorSize(CB);
  factor = ((double)VectorFreq(CB,p1) * (double)VectorFreq(CB,p2)) /
           ((double)VectorFreq(CB,p1) + (double)VectorFreq(CB,p2));
  dist_limit = roundpds((double)maxdist/factor);
  /* Precondition: Vsize >= 1 */
  /* if dist >= dist_limit, returns at least maxdist */
  do
        {
              diff = (llong)VectorScalar(CB,p1,i) -
                  (llong)VectorScalar(CB,p2,i);
              dist += diff * diff;
        } while(dist < dist_limit && ++i < Vsize);
  distortion=round( factor * (double)dist);

  return( distortion);
}
```