Efficient agglomerative clustering using *k* nearest neighbor graph

(submitted for publication 16.8.2004)

Pasi Fränti, Olli Virmajoki and Ville Hautamäki

Department of Computer Science
University of Joensuu
P.O. Box 111, FIN-80101 Joensuu, FINLAND
Email: franti@cs.joensuu.fi

Abstract: The search for nearest neighbor is the main source of computation workload in most clustering algorithms. A common operation is to calculate the distances to all candidates (full search) and select the one with the smallest distance. In this paper, we propose the use of nearest neighbor graph for reducing the number of candidates to be considered so that the number of distance calculations per search decreases. We apply the proposed scheme within agglomerative clustering algorithm, which is also known as the *PNN* algorithm, and show that remarkable reduction is obtained in the running time at the cost of slight increase in the distortion of the resulting clustering.

Keywords: Clustering, agglomeration, vector quantization, codebook generation, nearest neighbor graph, PNN.

Statistics: 25 pages, 14 figures, 10 tables, 9723 words, 48375 characters.

1. Introduction

Clustering is an important problem that must often be solved as a part of more complicated tasks in pattern recognition, image analysis and other fields of science and engineering [1, 2, 3, 4]. The clustering task is formalized here as a combinatorial optimization problem, in which the goal is to find the partition that minimizes a given distortion function.

Agglomerative clustering is a popular method for generating the clustering hierarchically by a sequence of merge operations. The clustering starts by initializing each data vector as its own cluster. Two clusters are merged at each step and the process is repeated until the desired number of clusters is obtained. *Ward's method* [5] selects the cluster pair to be merged that minimizes the increase in the distortion function value. In the vector quantization context, this method is known as the *pairwise nearest neighbor* (*PNN*) method due to [6].

Agglomerative clustering is an interesting method for clustering because of its conceptual simplicity and good results [7]. The method can also be combined with the *k-means* clustering such as the *generalized Lloyd algorithm* (GLA) [8] as proposed in [9], or used as a component in more sophisticated optimization methods. For example, agglomerative clustering has been used in the merge phase in the *split-and-merge* algorithm [10] resulting in to a good time-distortion performance, and as the crossover method in *genetic algorithm* [11], which has turned out to be the best clustering method among a wide variety of algorithms in terms of the quality of the codebook [12].

The main drawback of the agglomerative clustering is its slowness. The original implementation requires $O(N^3)$ distance calculations [13]. An order of magnitude faster algorithm has been introduced in [7] but the method is still lower bounded by $\Omega(N^2)$. The main source of computation originates from the search of the nearest neighbor cluster because the agglomerative clustering always calculates distances to all candidates when finding the nearest cluster.

Another approach for clustering is to use graph theoretical methods [1], [14], [15], [16], [17], [18], [19], [20]. For example, by first creating a complete undirected graph where the nodes correspond to the data vectors and the edges correspond to vector distances according to a given *similarity* or *dissimilarity* measure. The resulting graph can be trimmed to a *minimum spanning tree*, which can be interpreted as one large cluster. The clustering can then be generated iteratively by removing longest edges from the graph. In the final graph, clusters can be determined by finding the separate components in the graph [14]. This algorithm can be seen as a variation of split-based methods with similar criterion as in the *single-linkage* agglomerative clustering.

In this paper, we introduce fast agglomerative clustering algorithm motivated by the graph-based approaches. In our approach, we process the data at the cluster level so that every node in the graph represents a cluster and not as a single vector as in the previous approaches. The edges of the graph represent inter cluster connections between nearby clusters. The graph having linear space complexity is used merely as a search structure for reducing the number of distance calculations. In principle, we can apply any distortion function.

Many agglomerative clustering algorithms construct a sparse graph and then perform the clustering on this graph [1], [18], [19]. Two main differences between these and our approach are: (1) the existing methods construct an undirected graph while we construct a directed graph. (2) the existing methods neglect the original data after building the weighted graph (meaning that the weights of the new edges are determined by the weights of the current edges), but we still use the original data in order to compute the weights of newly formed edges as the agglomerative clustering goes on.

The proposed approach has two specific problems to solve: how to generate the graph efficiently, and how to utilize it. For example, standard solutions for determining a minimum spanning tree takes O(N^2) time, which would overweigh any speed-up. We propose solutions for the first problem by considering a *KD-tree* [21], *divide-and-conquer* [22], and *projection-based search* [23]. As to the second sub-problem, we analyze out how much speed-up can be gained by using the graph for reducing the number of calculations in the agglomerative clustering. Our experimental results indicates that a relatively small neighborhood size is sufficient for preserving good quality clustering. It is also possible that the idea could be generalized to other clustering algorithms that include large number of nearest neighbor searches.

The rest of the paper is organized as follows. In Section 2, we define the clustering problem considered here, and recall the method of agglomerative clustering. In section 3, we propose a new graph-based agglomerative clustering algorithm. Several solutions for creating the nearest neighbor graph are considered in Section 4. Experimental results are reported in Section 5, and conclusions drawn in Section 6.

2. Agglomerative clustering

The *clustering problem* is defined here as a combinatorial optimization problem. Given a set of N data vectors $X=\{x_1, x_2, \ldots, x_N\}$, partition the data set into M clusters so that a given distortion function is minimized. Partition $P=\{p_1, p_2, \ldots, p_N\}$ defines clustering by giving for each data vector the index of the cluster where it is assigned to. *A cluster* s_a is defined as the set of data vectors that belong to the same partition a:

$$s_a = \left\{ x_i \middle| p_i = a \right\}. \tag{1}$$

Clustering is then represented as the set $S=\{s_1, s_2, \ldots, s_M\}$. In vector quantization, the output of clustering is a codebook $C=\{c_1, c_2, \ldots, c_M\}$, which is usually the set of cluster centroids. We assume that the vectors belong to Euclidean space, and use the mean square error (*MSE*) as the distortion function:

$$MSE(C,P) = \frac{1}{N} \cdot \sum_{i=1}^{N} \left\| x_i - c_{p_i} \right\|^2. \tag{2}$$

The *agglomerative clustering* (*PNN*) [5, 6] generates clustering hierarchically using a sequence of merge operations. At each step two nearby clusters are merged:

$$s_a \leftarrow s_a \cup s_b. \tag{3}$$

The cost of merging two clusters s_a and s_b is the increase in the *MSE*-value caused by the merge. It can be calculated using the following formula [5, 6]:

$$d_{a,b} = \frac{n_a n_b}{n_a + n_b} \cdot \left\| c_a - c_b \right\|^2, \tag{4}$$

where n_a and n_b are the corresponding cluster sizes. The agglomerative clustering applies local optimization strategy: all possible cluster pairs are considered and the one increasing *MSE* least is chosen:

$$a,b = \arg \min_{\substack{i,j \in [1,m] \\ i \neq j}} d_{i,j}, \tag{5}$$

where m is the current number of clusters. There exist many variants of the agglomerative clustering. Straightforward implementation recalculates all distances at each step of the algorithm. This takes $O(N^3)$ time because there are $O(N)$ steps in total, and $O(N^2)$ cluster pairs to be checked at each step.

Another approach is to maintain an $N \times N$ matrix of the merge cost values. The merge cost values must be updated only for the newly merged cluster. Nevertheless, the algorithm still requires $O(N^3)$ because the search of the minimum cluster pair takes $O(N^2)$ time [13]. Kurita's method maintains an $N \times N$ matrix but it also utilizes a heap structure for searching the minimum distance [24]. The method runs in $O(N^2 \cdot \log N)$ time. The storage of the matrix, however, requires $O(N^2)$ memory, which makes these variants impractical for large data sets.

A fast implementation of the agglomerative clustering with linear memory consumption has been obtained by maintaining a pointer from each cluster to its nearest neighbor, and the corresponding merge cost value [7]. The cluster pair to be merged can be found in $O(N)$ time, and only a small number (denoted by τ) of the nearest neighbor needs to be updated after each merge. The implementation takes $O(\tau N^2)$ time in total. In the following, this method is called as *Fast PNN*.

Further speed-up can be achieved by using lazy update of the merge cost values [25], and by reducing the amount of work caused by the distance calculations [26]. It has been shown that in one-dimensional case (multilevel thresholding) agglomerative clustering can be performed efficiently in O(N·log$N$) time [27]. The result, however, does not generalize to higher dimensional data.

3. Agglomerative clustering using *kNN* graph

We define *k-nearest neighbor graph* (*kNN graph*) as a weighted directed graph, in which every node represents a single cluster, and the edges correspond to pointers to neighbor clusters. Every node has exactly k edges to the k nearest clusters according to a given distance function. The distance of clusters is defined by the merge cost function of Eq. (4). Note that this is not the only possible definition of the graph: other definitions have been given in [28], [29].

The proposed method is based on agglomerative clustering, but we utilize the graph structure in the search for the nearest neighbor clusters. In the agglomerative clustering, the search for the nearest neighbor cluster is repeated many times, and every search requires O(N) distance calculations. The graph is utilized so that the search is limited only to the clusters that are directly connected by the graph structure. This reduces the time complexity of every search from O(N) to O(k). If the number of edges (k) is small, significant speed-up can be obtained.

3.1 Simple implementation

The main structure of the algorithm is given in Fig. 1. The algorithm starts by initializing every data vector as its own clusters, and by constructing the neighborhood graph. The algorithm then iterates by removing nodes from the graph until the desired number of clusters has been reached. The graph stores the merge costs, i.e. the amount of extra distortion if the two neighbor clusters are merged. The edge cost values are stored in a heap structure.

At first, the edge with the smallest weight is found, and the nodes (s_a and s_b) are merged. The algorithm creates a new node s_{ab} from the clusters s_a and s_b, which are removed from the graph. The corresponding edges are updated by calculating new cost values between the nodes that were connected to the merged nodes. The algorithm must also calculate cost values for the outgoing edges from the newly created node s_{ab}. The k nearest neighbors are found among the $2k$ neighbors of the previously merged nodes s_a and s_b.

We illustrate the merge procedure in Fig. 2 when the clusters a and b are merged for a sample directed *2NN* graph (k=2). The nearest neighbors for the merged cluster is found among the neighbors of a and b. In this example, they are the clusters c and e. We must also update the links that pointed before either to the cluster a or to the cluster b to point to the new cluster and update the associated cost values. In practice, the new cluster replaces a, and b is removed. The pointers $c \rightarrow b$ and $d \rightarrow b$ are replaced by pointers $c \rightarrow a$ and $d \rightarrow a$, accordingly. A sample directed graph (k=4) is shown in Fig. 3.

```
AgglomerativeClustering(X, M) → S
    FOR i←1 to N DO
        s_i ← {x_i};
    FOR ∀(s_i; i ∈ [1, N]) DO
        Find k nearest neighbors;
    REPEAT
        (s_a, s_b) ← GetNearestClustersInGraph(S);
        s_ab ← Merge(s_a, s_b);
        Search the k nearest neighbors for s_ab;
        Update the nodes that had s_a and s_b as neighbors;
    UNTIL |S|=M;
```

Fig. 1. Structure of the proposed agglomerative clustering method.

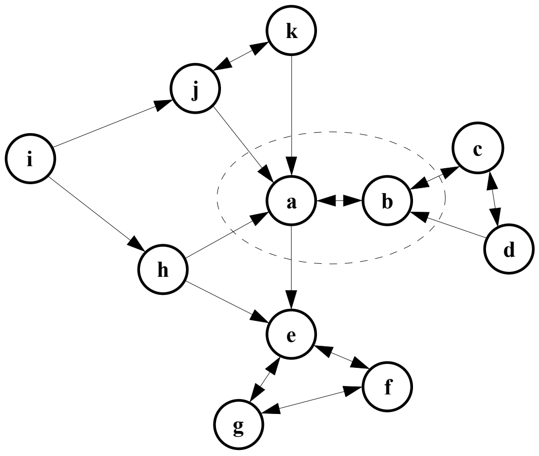


Fig. 2. Illustration of the graph (k=2) where a and b are to be merged.

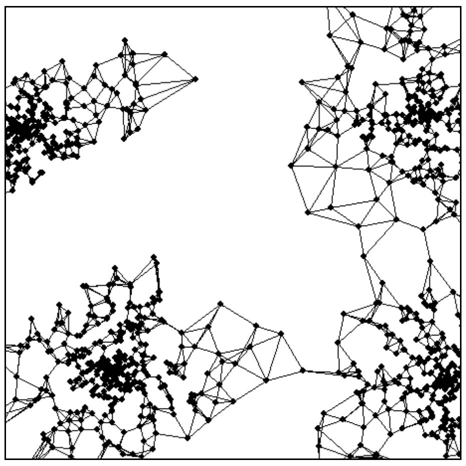


Fig. 3. Sample data set (black dots) and the corresponding directed graph (k=4). Note that the arrow heads are not printed for clarity.

3.2 Time complexity

The parameter k affects both the quality of the solution and the running time. It is therefore relevant for our complexity analysis to consider the extreme cases when k=N, and when k is a small constant. In the case when k=N, the algorithm produces exactly the same result as the full search agglomerative clustering. This claim is justified by the notion that when k=N we store all pairwise distances, and thus we can generate the same clustering as with the agglomerative clustering. The summary of the time complexity for the general case is presented in Table 1, where the "steps" means the total number of the steps in the loops per iteration, and "distances" means the total number of the distance operations calculated per iteration.

Each iteration of the algorithm takes O(1) time to get the smallest distance from the top of the heap structure (*Search nearest*). The merge of the two nearest clusters takes O($2k^2$ + log$N$) time (*Merge*). The first term ($2k^2$) originates from the selection of k neighbors by processing the neighbor list of the original clusters a and b separately. The second term (log$N$) comes from the update of the heap structure. We then have to find all the clusters that have the merged cluster as one of its k nearest neighbors (*Find neighbors*). This takes $O(kN)$ time. The removal of the last cluster takes $O(k + 2\log N)$ time (*Remove last*).

Finally, we update the distances of the incomings pointers into the heap structure, which takes $O(kN + (\tau/k)\cdot\log N)$ time (*Update distances*), where τ is the indegree of the node. There are N clusters to be checked, of which τ are to be considered, and only τ/k needs to be updated in the heap. It has been shown in [30] that in Euclidean space, τ is upper bounded by k times the *kissing number*, which is defined as number of unit hypespheres that are touching another unit hypersphere without any intersections. Kissing numbers are known for some dimensions, but unfortunately general formula is an open problem [31]. In practice, only a relatively small (about 2-4) number of neighbors must be updates in case of image data [7].

3.3 Double linked list

There are several ways to improve the simple implementation. Even if we consider k as a small constant, the time complexity of the algorithm sums up to $O(N^2 + \tau N\cdot\log N)$. In general, this is too much and we therefore consider the use of a double linked list as shown in Fig. 4. For every node, we maintain two lists: (1) the first list contains pointers to the k nearest clusters; (2) the second list contains τ "back pointers" to the clusters for which the current is one of their nearest neighbors. In this way, we can eliminate $O(kN)$ time loops, and the time complexity reduces to $O(\tau N\cdot\log N)$, see Table 1.

The observed number of steps and distance calculations are reported in Tables 2, 3 and 4 for three image data sets. The number of distance calculations can be reduced down to a fraction of that of the full search (from 40 million to 47 370 in the case of *Bridge*) by the simple implementation. However, the number of steps is reduced only to about 60-75% of that of the full search (from 81 million to 50 million in the case of *Bridge*). The double linked list also solves this problem (in the case of *Bridge*, the number of steps is reduced from 81 million to 517 905).

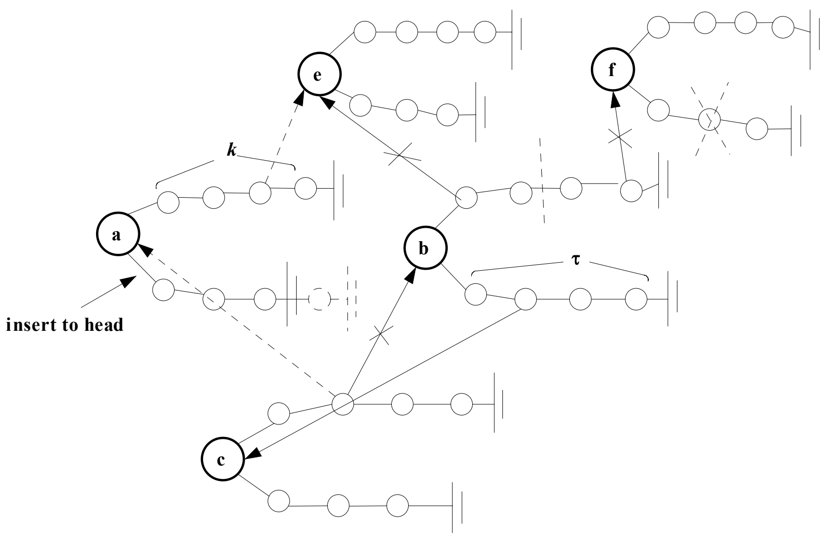


Fig. 4. Illustration of the update of the double linked list in the merge procedure of the clusters *a* and *b* in the neighborhood graph (k=4). The merged cluster is assigned to the place of the cluster *a* and finally the cluster *b* is removed.

Table 1: Estimated number of steps and distance calculations per iteration.

	Fast PNN		Proposed (simple)		Proposed (double-linked)	
	Steps	Distances	Steps	Distances	Steps	Distances
Search nearest	N	-	1	-	1	-
Merge	N	-	$2\,k^2 + \log N$	$2\,k$	$2\,k^2 + \tau k + \log N$	$2\,k$
Find neighbors	N	-	kN	-	τk	-
Remove last	N	-	$k + 2\log N$	-	$\log N$	-
Update distances	$N\,(1+\tau)$	τN	$kN + \tau/k \cdot \log N$	τ	$\tau k + \tau/k \cdot \log N$	τ

Table 2: Observed number of the steps and distance calculations per iteration for *Bridge* ($k = 3$).

	Fast PNN		Proposed (simple)		Proposed (double-linked)	
	Steps	Distances	Steps	Distances	Steps	Distances
Search nearest	8 357 760	-	3 840	-	3 840	-
Merge	8 357 760	-	100 636	13 302	181 159	13 316
Find neighbors	8 357 760	-	25 078 280	-	63 765	-
Remove last	8 349 185	-	94 779	-	45 514	-
Update distances	48 538 136	40 166 328	25 196 128	34 068	223 627	34 097
Total	81 960 601	40 166 328	50 468 663	47 370	517 905	47 413

Table 3: Observed number of the steps and distance calculations per iteration for *House* ($k = 3$).

	Fast PNN		Proposed (simple)		Proposed (double-linked)	
	Steps	Distances	Steps	Distances	Steps	Distances
Search nearest	581 798 432	-	33 856	-	33 856	-
Merge	581 798 432	-	960 410	112 923	1 245 180	112 942
Find neighbors	581 798 432	-	1 745 395 296	-	255 519	-
Remove last	580 354 971	-	1 026 580	-	497 511	-
Update distances	2 237 529 292	1 655 663 346	1 746 170 412	162 850	1 313 939	163 757
Total	4 563 279 559	1 655 663 346	3 493 586 554	275 773	3 346 005	276 699

Table 4: Observed number of the steps and distance calculations per iteration for *Miss America* ($k = 3$).

	Fast PNN		Proposed (simple)		Proposed (double-linked)	
	Steps	Distances	Steps	Distances	Steps	Distances
Search nearest	20 965 544	-	6 224	-	6 224	-
Merge	20 965 544	-	172 616	25 797	273 994	25 819
Find neighbors	20 965 544	-	62 896 632	-	81 537	-
Remove last	20 955 451	-	160 544	-	77 194	-
Update distances	128 322 309	107 331 780	63 078 024	44 418	326 677	44 331
Total	212 174 392	107 331 780	126 314 040	70 215	765 626	70 150

4. Creation of the neighborhood graph

Graph creation is related to the *post office* problem in computational geometry and in statistical pattern recognition, it is known as *kNN* classifier problem. The problem is to find for every data point x its closest vector from a smaller set of representative vectors. Many solutions exist (e.g. by solving minimum spanning tree for the special case k=1) at the cost of $O(N \cdot \log N + M^2)$, and with the assumption that all pairwise distances are stored in memory. In our case, however, we have M=N. This makes the time complexity to $O(N^2)$, which is the same as that of the full search.

Efficient construction of the k nearest neighbor graph is also needed in *manifold learning* [32], [33], [34]. It uses nonlinear dimensionality reduction by mapping the input vectors into a smaller dimensional subspace (called manifold). If such a manifold exists, the Euclidean distance can be assumed to hold between nearby vectors along the surface, which reduces the dimensionality significantly.

Other related problem in computational geometry is *k-all-nearest-neighbors* problem. The problem is to find for all vectors their k-nearest neighbors in the same set. Theoretical results can be found in [35], but we are not aware of any practical subquadratic algorithm that works for higher dimension K. In general, this is referred as the *curse of dimensionality*.

A straightforward solution is to construct the graph by *brute force* by considering all pairwise distances but at the cost of $O(N^2)$ time. In the following, we aim at solving the problem either by a faster but heuristic (thus suboptimal) method, or by allowing the worst case time complexity become $O(N^2)$ if the algorithm works faster for typical data sets in practice. We consider also the following three methods:

- *KD-tree* [21]
- *Divide-and-conquer* [22, 36]
- *Projection-based search* [23, 37]

For further information on recent algorithms for searching the nearest neighbor, see [38], [39].

4.1 *KD*-tree

There are many algorithms using spatial data structures for fast nearest neighbor search such as *KD-tree, ball-tree,* and *R-tree*. For example, *KD-tree* [21] has been introduced already long time ago and it is still widely used for finding the nearest neighbor [40]. These techniques can be extended for creating a k nearest neighbor graph.

The KD-tree is a generalization of a simple binary search tree, in which each node represents a subset of the vectors in the data set. The *root* of the tree represents the entire data set. Each non-terminal node has two children *sons* representing two subsets defined by the partitioning. The terminal nodes represent mutually exclusive small subset of the data sets, which collectively form a partition of the data set. These terminal subsets of vectors are called *buckets*. In K-dimensional vector space, a cluster is represented by K keys. Any of these can serve as the discriminator for partitioning the subset represented by a particular node in the tree. In the creation of the neighborhood graph, we insert every vector into the KD-tree, and then search for each vector its k nearest neighbors from the same tree.

The KD-tree data structure provides an efficient mechanism for examining only those vectors closest to the query vector, thereby greatly reducing the computation required to find the

k nearest neighbors. The search algorithm is most easily described as a recursive procedure. The geometric boundaries of the node are determined by the partitions defined at nodes above it in the tree. If the node under investigation is terminal, then all the vectors in the bucket are examined excluding the query vector in question. A list of k nearest neighbors so far found and their distances to the query vector is maintained as an ordered list during the search. Whenever a vector is examined and found to be closer than the most distant member of this list, the list is updated.

If the node under investigation is not terminal, the recursive procedure is called for the node representing the subset on the same side of the partition as the query vector. When the control returns, a test is made to determine if it is necessary to consider the vectors on the side of the partition opposite the query vector. This is referred to as the "*bounds-overlap-ball*" test. A "*ball-with-in-bounds*" test is made before returning to determine if it is necessary to continue the search. These tests are carried out with the distance function defined by the merge distortion function of the agglomerative clustering of Eq. (4). See [21] for details of the KD-tree algorithm.

The goal of the optimization of the KD-tree is to minimize the expected number of vectors examined with the search algorithm. The parameters to be adjusted are the discriminating key, partition value at each nonterminal node, and the number of vectors contained in each terminal bucket. The prescription for optimizing the KD-tree is to choose at every nonterminal node the key with the largest spread in values as discriminator and to choose the median of the discriminator key values as the partition.

The creation of the KD-tree takes $O(KN \cdot \log N)$ time and each search is proportional to $\log N$ [21]. Thus, if we consider K as a small constant, the expected time complexity of the algorithm is only $O(N \cdot \log N)$ in the case of the low dimensional data sets. Nearest neighbor search capabilities of KD-tree has been studied theoretically and experimentally by Yianilos [41]. It was concluded that for high dimensional uniformly distributed data, to achieve savings over exhaustive search, the search radii has to be very small. In other words, as dimensionality increases either the number of distance computations must be increased, or the search radii decreased.

4.2 Divide-and-conquer method

Closest pair problem [22] is stated as follows: given N points in K-dimensional space, find the two points whose mutual distance is minimal. The problem can be solved by divide-and-conquer technique as follows:

1. Divide X into X_1 and X_2 by the median hyperplane H normal to some axis.
2. Recursively solve the problem for X_1 and X_2.
3. Compute $\delta = \min(\delta_1, \delta_2)$, where δ_1 and δ_2 are the found distances in X_1 and X_2.
4. Let X_3 be the set of points that are within δ of H.
5. Recursively examine all pairs in X_3.

It has been shown that, in the case of 2-dimensional vector space, only a constant number of points can be neighbor in any cell in the set X_3 [42]. Assuming that the same primary axis is used in the division, the points can be pre-sorted and the analysis step can be performed in linear time. It has been proven that the algorithm takes $O(N \cdot \log N)$ time and the algorithm generalizes to multi-dimensional spaces but at the cost $O(N \cdot \log^{K-1} N)$ time [43], where K is the number of dimensions.

We consider next an algorithm applicable for finding *k*-near neighbors based on the above divide-and-conquer approach with the following differences [36]. Firstly, we search several closest pairs for every vector in the data set. Secondly, we use *principal component analysis* (PCA) for calculating the projection axis with the maximum deviation. Thirdly, we use a distance-based heuristic for selecting the vectors to be included in the third subset.

The pseudo code of the algorithm is given in Fig. 5. At each step of the recursion, we divide the data set X into two subsets X_1 and X_2 of equal sizes as follows. We first calculate the principal axis of the data vectors in X, and then select a (K-1)-dimensional hyperplane H perpendicular to the principal axis. The hyperplane is selected so that approximately half of the vectors belong to one side of the space, and the rest to the other side. Once the dividing procedure has been done, the two subproblems X_1 and X_2 are solved recursively. Subproblems smaller than c_k are solved by brute force search.

```
Divide-and-Conquer($X$, $k$, $c_k$ ) $\rightarrow$ $kNN$
    IF ( |X|  > $c_k$ ) THEN
        $X1$, $X2$, $proj \leftarrow$ Divide($X$);
        $kNN1 \leftarrow$ Divide-and-conquer($X1$, $k$, $c_k$);
        $kNN2 \leftarrow$ Divide-and-conquer($X2$, $k$, $c_k$);
        $kNN \leftarrow kNN1 \cup kNN2$;
        $X3 \leftarrow$ GenerateThirdSet($X$, $kNN$, $proj$);
        $kNN3 \leftarrow$ Divide-and-conquer($S3$, $k$, $c_k$);
        $kNN \leftarrow$ CombineResults($kNN$, $kNN3$);
    ELSE
        $kNN \leftarrow$ BruteForce($X$, $kNN$, $k$);
    END-IF
RETURN $kNN$;

GenerateThirdSet($X$, $kNN$, $proj$) $\rightarrow$ $X3$
    $X3 \leftarrow \emptyset$;
    FOR $i \leftarrow$ 1 TO $|X|$ DO
        $\delta \leftarrow$ ProjectionDistance($X[i]$, $proj$);
        IF $c\delta <$ kNN[i,1] THEN
            $X3 \leftarrow X3 \cup X[i]$;
RETURN $X3$;
```

Fig. 5. Sketch of the divide-and-conquer algorithm.

After the subproblems have been solved, we generate a third subset X_3 consisting of vectors that are closer to the dividing hyperplane H than to its nearest neighbor in the corresponding subset (X_1 or X_2). By using the control parameter c we can control the number of vectors chosen in the subset. Once the subset has been created, the algorithm is recursively applied to it. Finally, the results of the three subproblems are combined. In Fig. 6 we illustrate the division of the set X to three overlapping subsets (X_1, X_2, X_3) according to the dividing hyperplane H. The arrows indicate the nearest neighbors of the vectors.

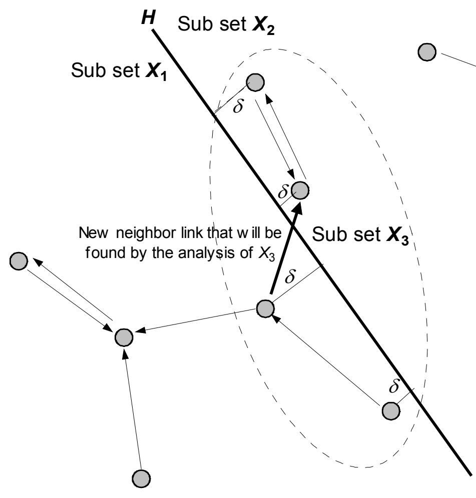


Fig. 6. Division to three overlapping subsets (X_1, X_2, X_3) according to the dividing hyperplane *H*. The arrows indicate the nearest neighbors of the vectors.

The time complexity of the proposed divide-and-conquer algorithm can be approximated by the recurrence $T(N) = 3\cdot T(N/2) + O(N\cdot K^2)$ assuming that the size of the third subset is less than equal to that of the other subsets X_1 and X_2. The second term originates from the calculation of the principal axis. The rest of the calculations can be performed in linear time. The recurrence solves to $O(K^2\cdot N^{1.58}\cdot \log N)$. It might be possible to squeeze the complexity by selecting the dividing hyperplane by some simpler method, and by making tighter bounds for the third subset. Note that the size of the X_3 is controlled by the parameter c, which can vary from $c=\infty$ (X_3 is empty) to $c=0$ ($X_3=X$). The time complexity of the first case ($c=\infty$) is $O(N\cdot \log N)$.

4.3 Projection-based search

Mean-distance ordered partial search (MPS) was originally proposed to be used with the *k-means* clustering (GLA) in [23] but then generalized to agglomerative clustering distance function in [26]. Here we apply it to the search for *k*-nearest neighbors as proposed in [37].

4.3.1. Searching for nearest neighbor

The method stores the component sums of each cluster centroid (code vector). Let s_a be the one, for which we seek its nearest neighbors, and s_j the candidate 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:

$$\hat{e}_{a,j} = \left(\sum_{k=1}^{K} c_{ak} - \sum_{k=1}^{K} c_{jk} \right)^2 . \tag{6}$$

The component sums correspond to the projections of the vectors to the diagonal axis of the vector space. In typical data 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. Then, given the cost function value of the best candidate found so far, vectors outside the radius defined by a given pre-condition can be excluded in the calculations, see Fig. 7.

In the agglomerative clustering, the cost function consists of the squared Euclidean distance $e_{a,j}$ of the code vectors (second part of Eq. 4), and the weighting factor $w_{a,j}$ (first part of Eq. 4), which can be calculated separately. The following inequality holds true (where K is the dimension of the data vector):

$$w_{a,j} \cdot \hat{e}_{a,j} \le K \cdot w_{a,j} \cdot e_{a,j} . \tag{7}$$

It was originally shown to hold in Euclidean distances in [23], which we have then generalized to the cluster distances in [26]. Given the cost function value d_{min} of the best candidate found so far, the inequality (7) can be utilized in the search of nearest neighbor by using the following *precondition*:

$$K \cdot d_{\min} < w_{a,j} \cdot \hat{e}_{a,j} . \tag{8}$$

In other words, if the squared Euclidean distance of the component sums (multiplied by the weighting factor) exceeds the distance to the best candidate found so far (multiplied by K), the value cannot be smaller than d_{min}, according to (7). This is illustrated in Fig. 7, where the distance from A to B is the current minimum. All potential candidates and their projections must therefore lie inside the circle.

The precondition is utilized as follows. The vectors are sorted according to their component sums, and then proceed in the order given by the sorting. The search starts from the cluster s_a and proceeds bidirectionally along the projection axis. The weighting factor $w_{a,j}$ and the distance of the component sums ($\hat{e}_{a,j}$) are first calculated, and the precondition (8) is evaluated. If it holds true, the calculation of the actual cost function value can be omitted and the candidate cluster s_j rejected. The precondition can be calculated fast in constant time as the component sums and weights are known.

In *k-means* clustering, the search in any of the two directions can be terminated immediately when the precondition is met first time. In the agglomerative clustering, however, this is not possible because of the weighting factor. Even in the initialization, there may be weighted vectors as the data set can be a result of preprocessing step where duplicate vectors have been merged and weighted by their frequency. The search is therefore terminated only if the weight of the candidate cluster equals to 1. See [26] for details.

The pseudo code of the algorithm is given in Fig. 8. For simplicity, we assume that the clusters have already been sorted before the call of the routine.

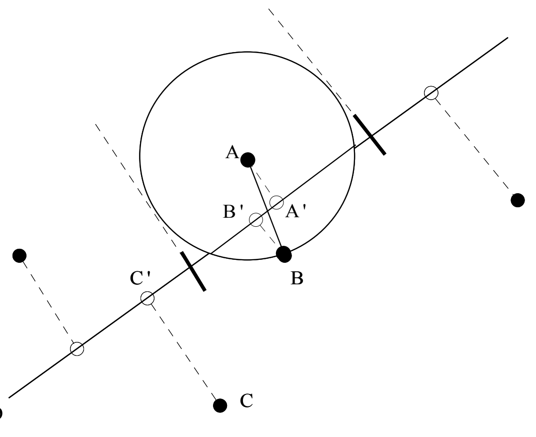


Fig. 7. Vectors (black dots) and their projections (white dots) according to component sums.

```
SearchNearestNeighborUsingMPS($s_a$, $S$) → $nn_a$, $d_a$;
    $d_{min} \leftarrow \infty$;
    up ← TRUE;
    down ← TRUE;
    $j_1 \leftarrow a$;
    $j_2 \leftarrow a$;

    WHILE (up OR down) DO
        IF up THEN
            $j_1 \leftarrow j_1 + 1$;
            IF $j_1 > N$ THEN up ← FALSE;
            ELSE CheckCandidate($s_a$, $s_{j1}$, $n_a$, $d_{min}$, $nn$, up);
        IF down THEN
            $j_2 \leftarrow j_2 - 1$;
            IF $j_2 < 1$ THEN down ← FALSE;
            ELSE CheckCandidate($s_a$, $s_{j2}$, $n_a$, $d_{min}$, $nn$, down);
    END-WHILE;
    RETURN $nn$, $d_{min}$;

CheckCandidate($s_a$, $s_j$, $n_a$, $d_{min}$, $nn$, direction) → $nn$, $d_{min}$;
    IF PreCondition($s_a$, $s_j$, $d_{min}$) THEN
        IF $n_a = 1$ THEN direction ← FALSE
    ELSE
        $d \leftarrow$ MergeCost($s_a$, $s_j$, $d_{min}$);
        IF $d < d_{min}$ THEN
            $d_{min} \leftarrow d$;
            $nn \leftarrow j$;
    RETURN $nn$, $d_{min}$;

PreCondition($s_a$, $s_j$, $d_{min}$) → BOOLEAN;
    $w \leftarrow n_a \cdot n_j / (n_a + n_j)$;
    $\hat{e} \leftarrow (sum_a - sum_j)^2$;
    RETURN( $K \cdot d_{min} < w \cdot \hat{e}$ );
```

Fig. 8. Pseudo code of the MPS method used for the graph creation. The input of the algorithm are the cluster s_a whose neighbor we are searching for and the entire clustering S. The output consists of the index and the distance of the nearest neighbor of the cluster s_a.

4.3.2 Searching for *k* neighbors

We apply the MPS method for finding the *k* nearest clusters as follows. We relax the condition of the graph and find any *k* neighbors instead of the nearest ones. This is a reasonable modification because the optimality of the graph cannot be guaranteed during the process of the agglomerative clustering. Thus, by relaxing the definition of the *k*-nearest neighbor graph, additional speed-up can be obtained at a slight increase in the distortion function value.

In particular, we use the exact MPS method for finding the nearest neighbor but stop the search immediately when it has been found. In addition to this, we maintain an ordered list of the *k* best candidates found so far. The rest of the neighbors are then chosen simply from the list of the candidates no matter whether they are actually the *k*-1 nearest or not. It is expected that the rest of the candidates are nearby vectors although not necessarily the nearest ones.

Even if some links were missing, vectors in the same cluster are most likely to be connected anyhow.

Another way to limit the search is to set up a fixed search range. In this case, the limit must be chosen experimentally. We will study these two alternatives (full search MPS and *limited search MPS*) later in Section 5.

The advantages of the MPS method are its simplicity and that it is expected to be fast on data sets with correlated vectors. The main disadvantage of the method is that the time complexity is still O(N^2), which is not any better than that of the *Brute force*. The actual speed-up is expected to be smaller on data sets with uncorrelated vectors.

4.3.3 Using PCA-projection

Instead of using component sum in the MPS method one can perform principal-component analysis (PCA) on the data set. Thus, one can use the projection to first principal component as the index in the purpose to speed up the search. When one has better projection axis the binary search for k nearest is probably terminated earlier. However, the calculation of the principal axis for projection using the *power method* takes O(NK^2) time. For high dimensional data sets it is likely that the extra work caused by the calculation of the principal axis exceeds the otherwise gained speed up.

4.4 Other methods

The other methods that can be considered for searching the k nearest neighbors include:

- *VPT* [44]
- *AESA* [39], [44]
- *MST* [14]
- *TIEC* [45]

Vantage point trees (*VPTs*) build a binary tree recursively, taking any vector p as the root and taking the *median M* of the set of all distances d. Those vectors u such that $d(p, u) \leq M$ are inserted into the left subtree, while those such that $d(p, u) > M$ are inserted into the right subtree. To solve a query in this tree, one measures $d{=}d(q, p)$. If $d\text{-}r \leq M$ the search enters into the left subtree, and if $d{+}r > M$ into the right subtree (with search radius r). One reports every vector considered that is close enough to the query. The *VPT* takes O(N) space and it is build in O($N\cdot\log N$) worst case time. The query complexity is argued to be O($\log N$), but as pointed out, this is true only for very small search radii, too small to be an interesting case.

Approximation elimination search algorithm (*AESA*) is experimentally shown to have O(1) query time. The structure is simply a matrix with the $N(N\text{-}1)/2$ precomputed distances among the vectors of the data set. At search time a vector p from the data set is selected at random and measure $r_\mathrm{p} = d(p, q)$, eliminating all vectors u of the data set that do not satisfy $r_\mathrm{p} - r \leq d(u, p) \leq r_\mathrm{p} + r$ (with search radius r). While all the $d(u, p)$ has been precomputed, so only $d(p, q)$ must be calculated at search time. The process of taking a random pivot among the (not yet eliminated) vectors of the data set and eliminating more vectors from the data set is repeated until the candidate set is empty and the query has been satisfied. See [34] for details of the algorithm. The problem with the algorithm is that it needs O(N^2) space and construction time which is unacceptably high for all but very small data sets.

Minimum spanning tree (*MST*) can be constructed in O($|E| + |V|\log|V|$) time, where $|E|$ is the number of the edges and $|V|$ is the number of the vertices. There is a data vector at each vertex of the *MST*, and the weight of an edge is the distance between the data vectors connected by the edge. In an MST, each vertex is always connected to at least one of its nearest neighbors. Thus, a partial ordering of the data vectors can be obtained. The *MST* presentation can be precomputed, and then stored in O(N) space. Unfortunately, in the case for searching the k nearest neighbors the precomputing takes O($N^2 + N\cdot\log N$) time.

The *triangle inequality elimination criteria* (*TIEC*) restrict the nearest neighbor search to a subsection of the data set based on the distances of the data vectors to an "anchor" vector. Given a fixed anchor vector, the distances between a anchor and each data vector is pre-computed and stored in O(N) space. Those distances serve as scalar projections of the data vectors with respect to the anchor vector. The use of several anchor vectors can strengthen *TIEC*'s ability to eliminate data vectors from consideration. However, each anchor vector requires O(N) space for the scalar projections, and the distances between each anchor vector and the other data vector needs to be calculated. If we use N anchors the *TIEC* take O(N^2) space and time.

5. Experiments

We consider three image data sets (Fig. 9), four synthetically generated data sets (Fig. 10), three *BIRCH* data sets [46], and six high dimensional data sets *Dim*032 to *Dim*1024. The vectors in the first set (*Bridge*) are 4×4 blocks taken from gray-scale image, and in the second set (*Miss America*) 4×4 difference blocks of two subsequent frames in video sequence. The third data set (*House*) consists of color values of the *RGB* image. The number of clusters is fixed to M=256. The data sets S_1 to S_4 are two-dimensional sets with varying complexity in terms of spatial data distributions with M=15 clusters. The data sets *Dim*032 to *Dim*1024 have slightly less spatial complexity but higher dimensionality varying from 32 to 1024 with M=256. The summary of the data sets is presented in Table 5. The algorithms are coded in *DJGPP C* Version 2.01 and are run on a 450 MHz Pentium III personal computer, in Microsoft Windows 98 Operating system.

Spatial vectors:	Spatial residual vectors:	Color vectors:
Bridge (256×256) K=16, N=4096	*Miss America* (360×288) K=16, N=6480	*House* (256×256) K=3, N=34112*

Fig. 9. Image data sets. *Duplicate training vectors are combined and frequency information is stored. Note that when duplicates vectors are merged, all distance and merge cost calculations must be multiplied by the frequency of the data vectors representing multiple instances of the original data set.

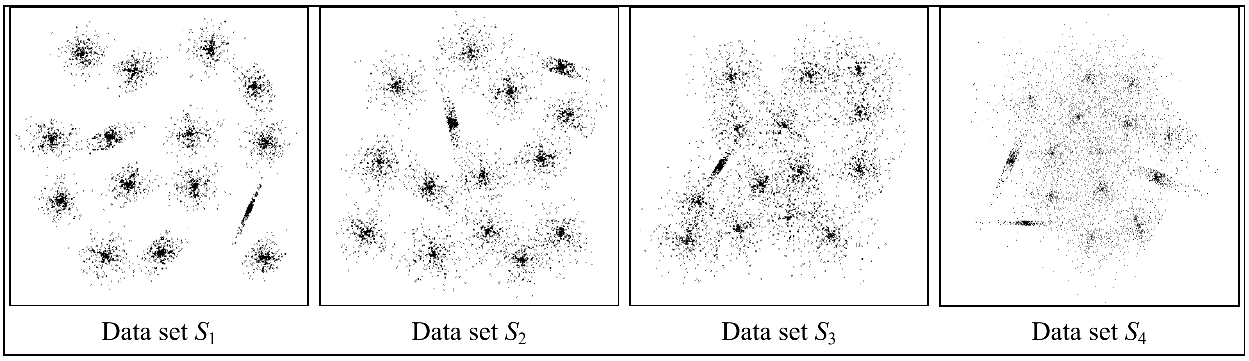


Fig. 10. Two-dimensional data sets with varying complexity in terms of spatial data distributions. The data sets have 5000 vectors scattered around 15 predefined clusters with a varying degree of overlap.

Table 5. Summary of the data sets.

Data set	Type of data set	Number of data vectors (N)	Number of clusters (M)	Dimension of data vector (K)
Bridge	Gray-scale image	4086	256	16
House	*RGB* image	34112	256	3
Miss America	Residual vectors	6480	256	16
Data set S_1- S_4	Synthetically generated	5000	15	2
BIRCH$_1$-$BIRCH_3$	Synthetically generated	100000	100	2
*Dim*32-1024	Synthetically generated	1000	256	32 – 1024

5.1 Parameter settings

The effect of the neighborhood size (parameter k) on the running time and quality is shown in Fig. 11 with the data sets *Bridge, House, Miss America* and $BIRCH_1$. The results indicate that a very small neighborhood size such as k=3 is sufficient for obtaining high quality clustering for the image data sets, and larger neighborhood sizes gives only slight improvements over these. In the case of noisy data with overlapping clusters (sets S_1 to S_4), however, too a small neighborhood size (k=3) can create isolated subclusters. A slightly larger neighborhood size (k=6) is therefore recommended for these data sets.

The running times of the agglomeration are summarized in Table 6 for the *simple* algorithm, and for the *double linked* algorithm in the case of the image data sets. The results show that the *simple* algorithm is useful with *Bridge* and *Miss America*, but not with the larger image set *House*. The *double linked* algorithm, on the other hand, works very fast ($\leq$1 second) in all cases. The MSE-values are virtually the same with both variants, as expected.

The overall running times and the corresponding number of distance calculations are summarized in Table 7. Comparative results are given for the *fast exact PNN* [7], and the *fast exact PNN with several speed-up methods* as proposed in [26]. The results show that the *proposed method* is significantly faster than the fast exact PNN with all data sets. The results are most remarkable with the largest data set (*House*), for which the running time was reduced down to 9 % from that of the fastest comparative variant. The corresponding numbers for *Bridge* and *Miss America* are 33 % and 43 %.

The running time has linear dependency with the parameter *k* but the growing rate is relatively small, see Fig. 11. The results also indicate that the graph creation is the bottleneck of the algorithm. We therefore study next the effect of the graph creation in more detail.

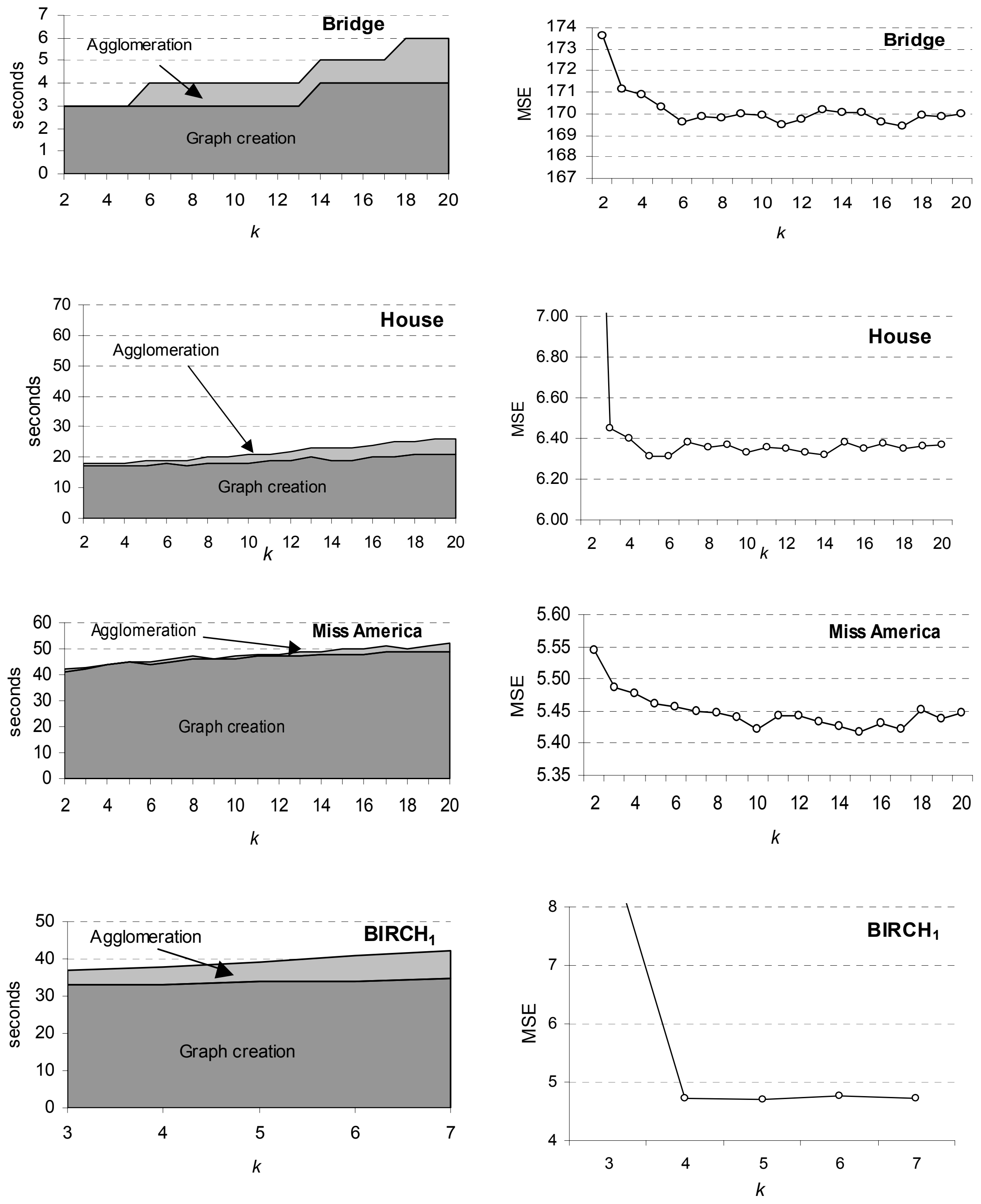


Fig. 11. The running time and quality of the proposed method as a function of *k*. The MPS algorithm is used for graph creation and the double linked list approach in the agglomeration.

Table 6. Running times of the iterations (excluding graph creation).

	Bridge		*House*		*Miss America*	
	Time	MSE	Time	MSE	Time	MSE
Simple	4	171.12	542	6.40	12	5.45
Double linked	< 1	171.11	1	6.37	< 1	5.44

Table 7. Summary of running times and the number of distance calculations of the proposed method (with MPS graph creation) in comparison to the best full search approaches [7, 26].

		Bridge		*House*		*Miss America*	
		Distance calculations	Run time	Distance calculations	Run time	Distance calculations	Run time
Fast PNN [7]		48 552 888	79	2 237 460 562	1574	128 323 740	229
Fast PNN + MPS + PDS + lazy [26]		6 167 439	9	37 752 863	190	83 323 889	106
Proposed	Graph creation	2 341 547	3	19 017 163	18	32 440 442	44
	Agglomerations	47 413	< 1	276 699	1	70 150	< 1
	Total	2 388 960	3	19 293 862	19	32 510 592	44

5.2 Graph creation

For the graph creation, we consider the following five algorithms from Section 4:

- *Brute force*
- *KD-tree*
- *Divide-and-conquer* (*D-n-C*)
- *Projection-based* (*MPS*)
- *Projection-based* (*MPS / PCA*)

Results for the image data sets are shown in Table 8, and for the *Dim* data sets in Table 9 using default parameter settings of the algorithms. For the image data sets, *divide-and-conquer* works best for the 16-dimensional data sets (*Bridge*, *Miss America*) whereas *KD-tree* is the most efficient for the 3-dimensional color vectors (*House*). The small differences in the MSE-values are due to the different order of processing.

For the higher dimensional data, the *MPS* method is the fastest while the *MPS / PCA* and the *Divide-and-conquer* becomes slower than the *Brute force* when the dimension grows high enough, see Fig. 12. These two methods calculate the principal axis for projection, which takes O(NK^2) time and thus, has quadratic dependency on the dimension. The *KD-tree* seems to operate reasonably fast without suffering of the curse of dimensionality. The reason might be that the clusters in these data sets are well separated and therefore having lower intrinsic dimension than representational dimension.

Overall, no method can be said to systematically outperform the others, and the graph creation remains the bottleneck of the algorithm in the sense running time. We therefore next fine-tune the methods towards faster running time at the cost of increasing distortion. We consider the following two variants:

- *Divide-and-conquer* by varying the parameter *c* (see Section 4.2),
- *Limited-search MPS* by setting a fixed search limit (see Section 4.3.2).

The corresponding time-distortion performance is illustrated in Fig. 13 for *House* and *Miss America*. Even though the results favor the divide-and-conquer method in the case of *Miss America*, it has much narrower operative time marginal, and the *limited-search MPS* is better in the case of all other sets. The KD-tree shows the best performance in the 3-dimensional data set *House*.

Table 8. Running times of the graph creation algorithms for the image data sets (*k*=3).

	Bridge		*House*		*Miss America*	
	Time	MSE	Time	MSE	Time	MSE
Brute force	34	171.17	881	6.43	89	5.44
KD-tree	12	170.42	5	6.36	79	5.44
D-n-C	2	171.80	49	6.58	7	5.44
MPS	3	171.11	18	6.37	44	5.44
MPS / PCA	10	170.97	37	6.39	58	5.43

Table 9. Running times of the graph creation algorithms for the *Dim* data sets (*k*=3).

	*Dim*032		*Dim*064		*Dim*128		*Dim*256		*Dim*512		*Dim*1024	
	Time	MSE	Time	MSE	Time	MSE	Time	MSE	Time	MSE	Time	MSE
Brute force	4	2.16	7	0.93	15	0.57	29	0.35	57	0.23	113	0.14
KD-tree	< 1	2.16	1	0.93	3	0.57	4	0.35	8	0.23	15	0.14
D-n-C	1	2.16	2	0.93	7	0.57	28	0.35	142	0.23	426	0.14
MPS	1	2.16	1	0.93	1	0.57	1	0.35	3	0.23	5	0.14
MPS / PCA	1	2.16	1	0.93	2	0.57	9	0.35	42	0.23	149	0.14

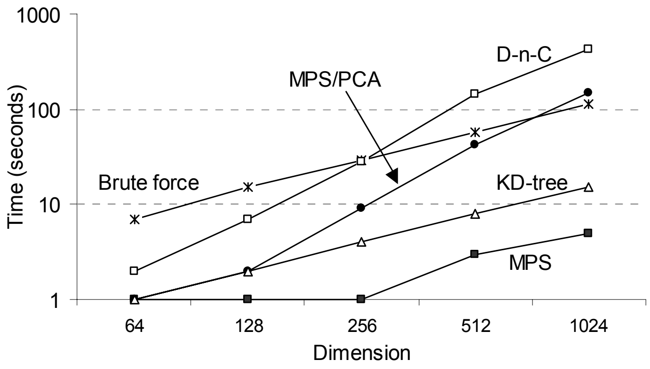
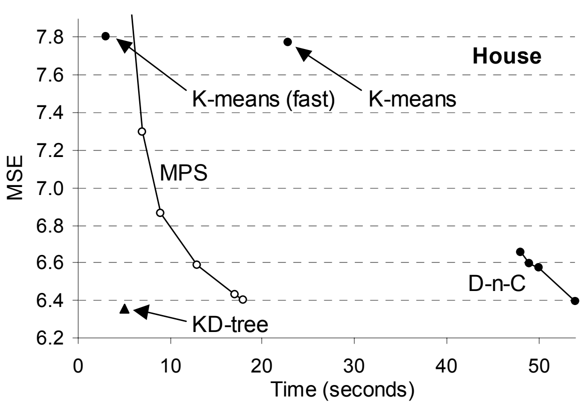


Fig. 12. Running times of the graph creation versus the dimension of the data vector (*k*=3).



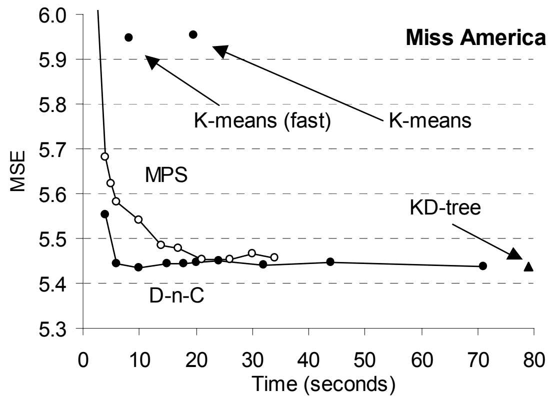


Fig. 13. Time-distortion performance of the proposed method (*k*=3).

5.3 Comparison

Finally, the proposed method is compared against the best full search PNN variants, and against standard *k*-means algorithm. Comparative results are summarized in Table 10 for all data sets with the following methods included:

- *Fast PNN* [7]
- *Proposed*
- *Proposed + k-means*
- *K-means* [8]

The *Fast PNN* has two variants: the fast implementation as proposed in [7] and the improved variant [26]. The latter one uses three speed-up techniques: *PDS*, *MPS* and *Lazy evaluation* of the distances. The *k-means* has two variants: the original method [8], and a faster variant, which uses PDS, MPS and activity detection for speed-up [47]. Results are given also for the *proposed method + k-means*, in which the data is first processed by the *proposed method* and the result is input to the *k-means*. The results show that the *proposed method* produces better result with an algorithm that is competitive to the *k-means* in speed.

The final clustering and the neighborhood graph of the *proposed method* are illustrated in Fig. 14 for the data set S_2. It shows that the *proposed method* achieves the correct clustering whereas the k-means fails to locate the clusters properly. Note that the final graph does not always have *k*=5 outgoing edges for every node although every cluster is still connected by the graph. With a smaller number of neighbors (*k*=3), however, there would have been isolated components and, in some cases, the algorithm degenerated to situation where there were not enough edges to achieve the final clustering. It is therefore recommended to use slightly larger neighborhood size, just in case.

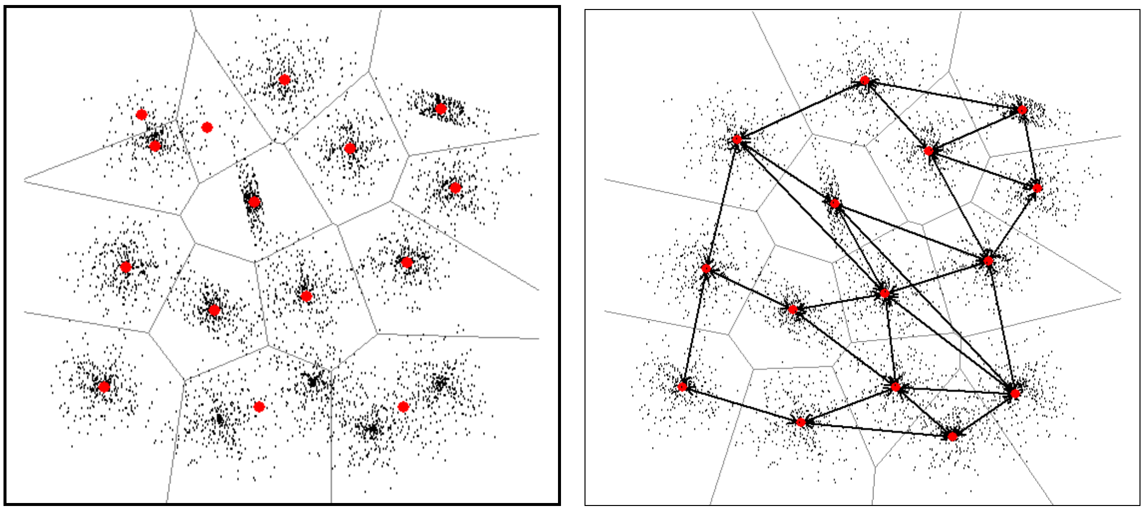


Fig. 14. Clustering of set S_2 by *k-means* (left) showing the cluster centroids and the corresponding Voronoi partition of the space; clustering by the proposed method using *k*=5 (right) showing the cluster centroids and the remaining neighborhood links.

Table 10. Comparison of the proposed method (k=5) with the existing algorithms.

Image data sets		*Bridge*		*House*		*Miss America*	
		Time	MSE	Time	MSE	Time	MSE
Fast PNN	Full search	79	168.92	1574	6.27	229	5.36
	+PDS+MPS+Lazy	9	168.92	190	6.26	106	5.37
Proposed	Full MPS	3	170.28	19	6.33	45	5.41
	Limited search MPS	3	170.56	14	6.51	6	5.58
Proposed + k-means	Full MPS	4	166.23	20	6.14	47	5.30
	Limited search MPS	4	166.38	15	6.18	9	5.34
K-means	Standard	13	179.95	23	7.77	20	5.95
	+PDS+MPS+Activity	2	180.02	3	7.80	8	5.95

Birch data sets		*$BIRCH_1$*		*$BIRCH_2$*		*$BIRCH_3$*	
		Time	MSE	Time	MSE	Time	MSE
Fast PNN	Full search	> 9999	4.73	> 9999	2.28	> 9999	1.96
	+PDS+MPS+Lazy	2397	4.73	2115	2.28	2316	1.96
Proposed	Full MPS	40	4.71	16	2.28	34	1.96
	Limited search MPS	37	4.73	15	2.28	28	2.02
Proposed + k-means	Full MPS	44	4.64	17	2.28	51	1.87
	Limited search MPS	41	4.64	16	2.28	44	1.90
K-means	Standard	209	5.51	43	7.42	171	2.41
	+PDS+MPS+Activity	29	5.34	8	7.85	35	2.50

Synthetic data sets		*S_1*		*S_2*		*S_3*		*S_4*	
		Time	MSE	Time	MSE	Time	MSE	Time	MSE
Fast PNN	Full search	25	8.93	25	13.44	25	17.70	25	17.52
	+PDS+MPS+Lazy	3	8.93	3	13.44	3	17.70	3	17.52
Proposed	Full MPS	< 1	9.07	< 1	13.41	< 1	17.21	< 1	16.69
	Limited search MPS	< 1	9.07	< 1	13.41	< 1	17.21	< 1	16.69
Proposed + k-means	Full MPS	< 1	8.92	< 1	13.28	< 1	16.89	< 1	15.71
	Limited search MPS	< 1	8.92	< 1	13.28	< 1	16.89	< 1	15.71
K-means	Standard	< 1	19.02	< 1	18.78	< 1	19.78	< 1	16.72
	+PDS+MPS+Activity	< 1	18.07	< 1	16.69	< 1	18.53	< 1	16.71

6. Conclusions

Fast agglomerative clustering using *k* nearest neighbor graph was proposed. A relatively small neighborhood size is sufficient to produce clustering with similar quality to that of the full search. At the same time, significantly fewer distance calculations and operations are needed and, therefore, remarkable speed-up is achieved. The running time is comparable to that of the *k*-means with a lower distortion.

Several algorithms for graph creation algorithms are considered, of which the projection-based heuristic (MPS) works reasonably well in most cases. The divide-and-conquer is faster in the case of some high dimensional image data sets, and the KD-tree in the case of 3-dimensional color clustering.

The proposed method has also some weaknesses. First of all, the graph creation is the bottleneck of the algorithm. It remains an open question whether faster method could be invented with better time-distortion performance than the proposed divide-and-conquer and the limited-search MPS algorithm.

Secondly, no theoretical solutions are given to set-up the neighborhood size others than experimentally. Fortunately, no major complications arose and the results were consistently better than that of the *k*-means. However, the method was detected to fail in a similar manner as the *k*-means if the neighborhood size is set too low. One possible solution would be to use a variable-size neighborhood depending on the overall distance distribution, or to apply multi-resolution approach in order to guarantee the connectivity of the graph overall.

To sum up, we conclude that the improvement due to the neighborhood graph is significant. The idea could also be applicable in other clustering algorithms also. This is a topic for future research.

References

[1] A.K. Jain and R.C. Dubes, *Algorithms for Clustering Data*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[2] B.S. Everitt, *Cluster Analysis* (3rd edition). Edward Arnold / Halsted Press, London, 1992.

[3] A. Gersho and R.M. Gray, *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Dordrecht, 1992.

[4] L. Kaufman and P.J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley Sons, New York, 1990.

[5] J.H. Ward, "Hierarchical grouping to optimize an objective function," *J. Amer. Statist.Assoc.*, 58, pp. 236-244, March 1963.

[6] W.H. Equitz, "A new vector quantization clustering algorithm," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 37 (10), pp. 1568-1575, October 1989.

[7] P. Fränti, T. Kaukoranta, D.-F. Shen and K.-S. Chang, "Fast and memory efficient implementation of the exact PNN," *IEEE Trans. on Image Processing*, 9 (5), pp. 773-777, May 2000.

[8] Y. Linde, A. Buzo and R.M. Gray, "An algorithm for vector quantizer design," *IEEE Trans. on Communications*, 28 (1), pp. 84-95, January 1980.

[9] 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 Trans. on Circuits and Systems for Video Technology*, 5 (2), pp. 83-95, April 1995.

[10] T. Kaukoranta, P. Fränti and O. Nevalainen, "Iterative split-and-merge algorithm for VQ codebook generation," *Optical Engineering*, 37 (10), pp. 2726-2732, October 1998.

[11] P. Fränti, J. Kivijärvi, T. Kaukoranta and O. Nevalainen, "Genetic algorithms for large scale clustering problem," *The Computer Journal*, 40 (9), pp. 547-554, 1997.

[12] P. Fränti, "Genetic algorithm with deterministic crossover for vector quantization," *Pattern Recognition Letters*, 21 (1), pp. 61-68, January 2000.

[13] J. Shanbehzadeh and P.O. Ogunbona, "On the computational complexity of the LBG and PNN algorithms," *IEEE Trans. on Image Processing*, 6 (4), pp. 614-616, April 1997.

[14] J.C. Gover and G.J.S. Ross, "Minimum spanning trees and single linkage cluster analysis," *Applied Statistics*, 18, pp. 54-64, 1969.

[15] E. Hartuv and R. Shamir, "A clustering algorithm based on graph connectivity," *Information Processing Letters*, 76 (4-6), pp. 175-181, December 2000.

[16] G.C. Osbourn and R.F. Martinez, "Empirically defined regions of influence for cluster analysis," *Pattern Recognition*, 28 (11), pp. 1793-1806, November 1995.

[17] S. Bandyopadhyay, "An automatic shape independent clustering technique," *Pattern Recognition*, 37 (1), pp. 33-45, January 2004.

[18] G. Karypis, E. Han and V. Kumar, "CHAMELEON: A hierarchical clustering algorithm using dynamic modeling," *IEEE Computer*, 32 (8), pp. 66-75, August 1999.

[19] D. Harel and Y. Koren, "Clustering spatial data using random walks," *The 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD'01)*, San Francisco, California, USA, pp. 281-286, August 2001.

[20] M.R. Brito, E.L. Chavez, A.J. Quiroz and J.E. Yukich, "Connectivity of the mutual *k*-nearest-neighbor graph in clustering and outlier detection," *Statistics & Probability Letters*, 35 (1), pp. 33-42, August 1997.

[21] J.L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, 18 (9), pp. 509-517, September 1975.

[22] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[23] S.-W. Ra and J.K. Kim, “A fast mean-distance-ordered partial codebook search algorithm for image vector quantization,” *IEEE Trans. on Circuits and Systems*, 40 (9), pp. 576-579, September 1993.

[24] T. Kurita, "An efficient agglomerative clustering algorithm using a heap," *Pattern Recognition*, 24 (3), pp. 205-209, March 1991.

[25] T. Kaukoranta, P. Fränti and O. Nevalainen, "Vector quantization by lazy pairwise nearest neighbor method," *Optical Engineering*, 38 (11), pp. 1862-1868, November 1999.

[26] O. Virmajoki, P. Fränti and T. Kaukoranta, "Practical methods for speeding-up the pairwise nearest neighbor method," *Optical Engineering*, 40 (11), pp. 2495-2504, November 2001.

[27] O. Virmajoki and P. Fränti, "Fast pairwise nearest neighbor based algorithm for multilevel thresholding," *Journal of Electronic Imaging*, 12 (4), pp. 648-659, October 2003.

[28] S. Arya and D.M. Mount, "Algorithm for fast vector quantization," *Proceedings of Data Compression Conference*, Snowbird, Utah, pp. 381-390, 1993.

[29] A.D. Constantinou, R.D. Bull and C.N. Canagarajah, "A new class of VQ codebook design algorithms using adjacency maps," *SPIE Electronics Imaging 2000*, San Jose, 3974, pp. 625-634, 2000.

[30] G.L. Miller, S-H. Teng, W. Thurston and S.A. Vavaris, "Separators for sphere-packings and nearest neighbor graphs," *Journal of the ACM*, 44 (1), pp. 1-29, January 1997.

[31] J.H. Conway and N.J.A. Sloane, *Sphere Packings, Lattices and Groups*, Springer-Verlag, New York, 1998.

[32] J.B. Tenenbaum, V. de Silva and J.C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science*, 290 (5500), pp. 2319-2323, December 2000.

[33] L.K. Saul and S.T. Roweis, "Think globally, fit locally: unsupervised learning of low dimensional manifolds," *Journal of Machine Learning Research*, 4, pp. 119-155, June 2003.

[34] M. Belkin and P. Niyogi, "Laplacian eigenmaps for dimensionality reduction and data representation," *Neural Computation*, 15 (6), pp. 1373-1396, June 2003.

[35] P.B. Callahan and S.R. Kosaraju, "A decomposition of multidimensional point sets with applications to *k*-nearest-neighbors and *n*-body potential fields," *Journal of the Association for Computing Machinery*, 42 (1), pp. 67-90, January 1995.

[36] O. Virmajoki and P. Fränti, "Divide-and-conquer algorithm for creating neighborhood graph for clustering," *Int. Conf. on Pattern Recognition (ICPR'04*), Cambridge, UK, 1, pp. 264-267, August 2004.

[37] P. Fränti, O. Virmajoki and V. Hautamäki, "Fast PNN-based clustering using *k*-nearest neighbor graph," *IEEE Int. Conf. on Data Mining* (*ICDM'03*), Melbourne, Florida, USA, pp. 525-528, November 2003.

[38] V. Ramanasubramanian and K.K. Paliwal, "Fast nearest neighbor search algorithm based on approximation-elimination search," *Pattern Recognition*, 33 (9), pp. 1497-1510, September 2000.

[39] S. Bandyopadhyay and U. Maulik, "Efficient prototype reordering in nearest neighbor classification," *Pattern Recognition*, 35 (12), pp. 2791-2799, December 2002.

[40] T. Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman and A.Y. Wu, "An efficient *k*-means clustering algorithm: analysis and implementation," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 24 (7), pp. 881-892, July 2002.

[41] P.N. Yianilos, "Locally lifting the curse of dimensionality for nearest neighbor search," *Proceedings of the Eleventh ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, San Francisco, California, USA, pp. 361-370, January 2000.

[42] M.I. Shamos, "Geometric complexity," *Proc. 7th Annual ACM Symposium on the Theory of Computing*, pp. 224-233, Albuquerque, New Mexico, 1975.

[43] J.L. Bentley and M.I. Shamos, "Divide-and-conquer in multidimensional space," *Proceedings of the 8th Annual ACM Symposium on the Theory of Computing*, pp. 220-230, Hershey, PA, May 1976.

[44] E. Chávez, G. Navarro, R. Baeza-Yates and J.L. Marroquín, "Searching in metric spaces," *ACM Computing Surveys*, 33 (3), pp. 273-321, September 2001.

[45] W. Li and E. Salari, "A fast vector quantization encoding method for image compression," *IEEE Trans. on Circuit and Systems for Video Technology*, 5 (2), pp. 119-123, April 1995.

[46] T. Zhang, R. Ramakrishnan and M. Livny, "BIRCH: A new data clustering algorithm and its applications," *Data Mining and Knowledge Discovery*, 1 (2), pp. 141-182, June 1997.

[47] T. Kaukoranta, P. Fränti and O. Nevalainen, "A fast exact GLA based on code vector activity detection," *IEEE Trans. on Image Processing*, 9 (8), pp. 1337-1342, August 2000.