Fast Agglomerative Clustering Using a k-Nearest Neighbor Graph

Pasi Fränti, Olli Virmajoki, and Ville Hautamäki

Abstract—We propose a fast agglomerative clustering method using an approximate nearest neighbor graph for reducing the number of distance calculations. The time complexity of the algorithm is improved from $O(\tau N^2)$ to $O(\tau N \log N)$ at the cost of a slight increase in distortion; here, τ denotes the number of nearest neighbor updates required at each iteration. According to the experiments, a relatively small neighborhood size is sufficient to maintain the quality close to that of the full search.

Index Terms—Clustering, agglomeration, nearest neighbor, vector quantization, PNN.

1 Introduction

THE *agglomerative clustering* method [1] is commonly used for clustering because of its conceptual simplicity. Its main drawback is its slowness, as the original implementation requires $O(N^3)$ time [2]. An order of magnitude faster algorithm has been introduced in [3], but the method is still lower bounded by $\Omega(N^2)$. The main source of computation originates from the search of the nearest neighbor.

Another approach is to use graph theoretical methods [4], [5]. For example, by first creating a complete undirected graph, where the nodes correspond to the data vectors. The graph can be trimmed to a *minimal spanning tree* representing one large cluster, which is then iteratively split by removing largest edges one by one. The final clustering is then determined by finding the separated components [4]. This is a variation of divisive clustering with similar criteria as in the *single-linkage* agglomerative clustering [6]. For agglomerative clustering, graph theoretical methods have also been used by constructing a sparse graph and then performing the clustering on this graph [7], [8].

We introduce a fast agglomerative clustering algorithm motivated by the graph-based approaches. However, we process the data so that every node represents a cluster and not a single vector. The main difference is that the existing methods neglect the original data after building the weighted graph, whereas we use the original data in order to compute the weights of newly formed edges as the agglomerative clustering goes on. We use the graph merely as a search structure for reducing the number of distance calculations.

The proposed approach has two specific problems: how to generate the graph efficiently, and how to utilize it. For example, standard solutions for solving the minimum spanning tree take $O(N^2)$ time, which would overweigh any speedup. For the first problem, we consider *K-d tree* [9], *divide-and-conquer* [10], and *projection-based search* [11]. For the second problem, we consider a double linked list for utilizing the graph structure. Experiments show that a significant speedup can be achieved and a relatively small neighborhood size is sufficient for preserving the quality of the clustering.

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

$$s_a = \{x_i | p_i = a\}. \quad (1)$$

The clustering is then represented as the set $S = \{s_1, s_2, \ldots, s_M\}$. In vector quantization, the output of the 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:

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

The *agglomerative clustering* method [1], [12] generates the clustering hierarchically using a sequence of merge operations. At each iteration, two nearby clusters are merged:

$$s_a \leftarrow s_a \cup s_b. \quad (3)$$

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 [1], [12]:

$$MergeCost(a, b) = \frac{n_a n_b}{n_a + n_b} \cdot \|c_a - c_b\|^2, \quad (4)$$

where n_a and n_b are the corresponding cluster sizes. This will be later denoted as the distance of the clusters a and b. *Ward's method* [1] selects the cluster pair to be merged that minimizes the increase in the distortion function value:

$$a, b = \arg\min_{\substack{i,j \in [1,m] \\ i \neq j}} MergeCost(i, j), \quad (5)$$

where m is the current number of clusters. In the vector quantization context, this is known as the *pairwise nearest neighbor (PNN)* method due to [12]. Straightforward implementation recalculates all distances at each iteration of the algorithm. This takes $O(N^3)$ time because there are $O(N)$ iterations, and $O(N^2)$ cluster pairs to be checked at each iteration.

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 [2]. Kurita's method maintains an $N \times N$ matrix, but it also utilizes a heap structure for searching the minimum distance [13].

• *The authors are with the Speech and Image Processing Unit, Department of Computer Science, University of Joensuu, PO Box 111, FIN-80101 Joensuu, Finland. E-mail: {franti, ovirma, villeh}@cs.joensuu.fi.*

Manuscript received 3 Oct. 2005; revised 23 Mar. 2006; accepted 3 Apr. 2006; published online 14 Sept. 2006.
Recommended for acceptance by L. Kuncheva.
For information on obtaining reprints of this article, please send e-mail to: tpami@computer.org, and reference IEEECS Log Number TPAMI-0534-1005.

AgglomerativeClustering(X, M) $\rightarrow$ S

Step 1: Construct k-NN graph.
Step 2: REPEAT
Step 2.1: Find the best pair (s_a, s_b) to be merged.
Step 2.2: Merge the pair (s_a, s_b) $\rightarrow$ s_{ab}.
Resolve k-NN list for s_{ab}.
Step 2.3: Remove obsolete (s_b).
Step 2.4: Find neighbors of s_{ab}.
Step 2.5: Update distances for the neighbors of s_{ab}.
UNTIL $|S|=M$;

Fig. 1. The structure of the proposed algorithm.

The method runs in $\mathrm{O}(N^2 \cdot \log N)$ time, but the matrix requires $\mathrm{O}(N^2)$ memory.

A fast implementation with linear memory consumption maintains a pointer from each cluster to its nearest neighbor [3]. The cluster pair to be merged can be found in $\mathrm{O}(N)$ time and only a small number (denoted by τ) of the nearest neighbors need to be updated after each merge. The implementation takes $\mathrm{O}(\tau N^2)$ time in total. We refer to this as the *fast exact PNN*.

Further speedup can be achieved by using lazy update of the merge cost values [14] and by reducing the amount of work caused by the distance calculations [15]. It has been shown that in a one-dimensional case (multilevel thresholding), agglomerative clustering can be performed in $\mathrm{O}(N \cdot \log N)$ time [16]. The result, however, does not generalize to higher dimensional data.

An inexact $\mathrm{O}(N \cdot \log N)$ time variant has also been considered in [12] by using *K-d* trees for localizing the search for the clusters and by merging several cluster pairs during the same iteration. This variant can be very fast, but it significantly decreases the quality of the clustering. Nevertheless, the idea itself of limiting the search of the nearest cluster is appropriate, and we continue the work in the same direction.

3 Agglomerative Clustering Using k-NN Graphs

We define a *k-nearest neighbor graph (k-NN graph)* as a weighted directed graph, in which every node represents a single cluster and the edges represent pointers to neighbor clusters. Every node has exactly k edges to the k nearest clusters, according to (4).

In agglomerative clustering, the search for the nearest neighbor is repeated several times per iteration and every search requires $\mathrm{O}(N)$ merge cost 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 $\mathrm{O}(N)$ to $\mathrm{O}(k)$. The parameter k affects the quality of the solution and the running time. If the number of neighbors (k) is small, significant speedup can be obtained.

3.1 The Simple Algorithm

The main structure of the algorithm is given in Fig. 1. The algorithm starts by constructing the neighborhood graph in the first step and, then, iteratively merges pairs of clusters until the desired number of clusters has been reached.

At Step 2.1, the edge with the smallest weight is found. For each cluster, we store the smallest merge cost value associated to the edge in a *heap structure* [17]. It is a binary tree, in which the minimum value is stored in the root (top of heap) and the values of the children are always greater than that of the parent node. In this way, we always find the best pair to be merged from the top of the heap and the heap structure can be updated in logarithmic time after the changes in the merge cost values. Note that only the nearest of the k-NN pointers is stored for each cluster.

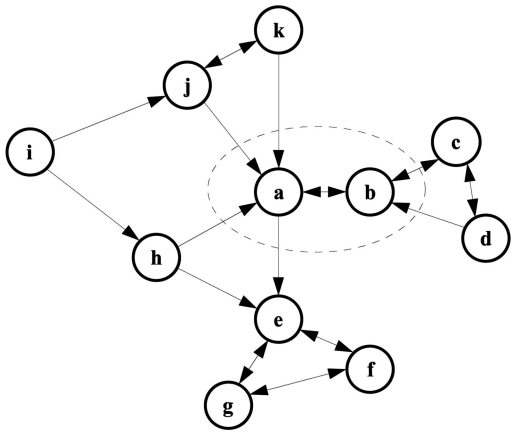


Fig. 2. An illustration of the k-NN graph $(k = 2)$ where a and b are to be merged.

At Step 2.2, the nodes (s_a and s_b) are merged and a new k-NN list is constructed for s_{ab}. In order to keep the computation reasonable, we select the k nearest neighbors from the $2k$ neighbors of the previously merged nodes s_a and s_b. This also means that the accuracy of the k-NN graph is compromised and, thus, the graph becomes an approximated nearest neighbor graph. It may also happen that the number of neighbors for the cluster s_{ab} can become smaller than k. The merged node (s_{ab}) replaces s_a and the second cluster (s_b) is removed from the data structures at step 2.3. At Step 2.4, we find the neighbors that pointed to s_a or s_b, and we update the corresponding merge cost values at Step 2.5.

We illustrate the procedure in Fig. 2 for a sample directed 2-NN graph ($k = 2$), where a and b are merged. The new k-NN list of the merged cluster is found among the neighbors of a and b, which are c and e. We also update the edges that pointed to the clusters a and b to point to the new cluster and update the associated cost values. The corresponding locations in the heap structure must also be updated. 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 graph ($k = 4$) is shown in Fig. 3.

3.2 Time Complexity

The summary of the time complexities are presented in Table 1, where the "steps" indicate the number of times the loops are performed per iteration and the "distances" indicate the number of distance calculations (4). We can consider k as a small constant, whereas in the extreme case ($k = N$), the algorithm would produce exactly the same result as the full search.

At each iteration, the best pair can be found from the heap in $\mathrm{O}(1)$ time (*Find the best pair*). The merge takes $\mathrm{O}(k^2 + \log N)$ time (*Merge*). The first term (k^2) comes from finding the k neighbors from the k-NN lists of a and b. In principle, we could merge the two lists in $\mathrm{O}(k)$ time, but as the merge changes the weights, the lists have no proper ordering anymore after merging and we apply insertion sort requiring $\mathrm{O}(k^2)$ time. The second term ($\log N$) comes from the update of the heap structure.

We then find all the clusters that have the merged one in their k-NN list (*Find neighbors*). This takes $\mathrm{O}(kN)$ time because the only

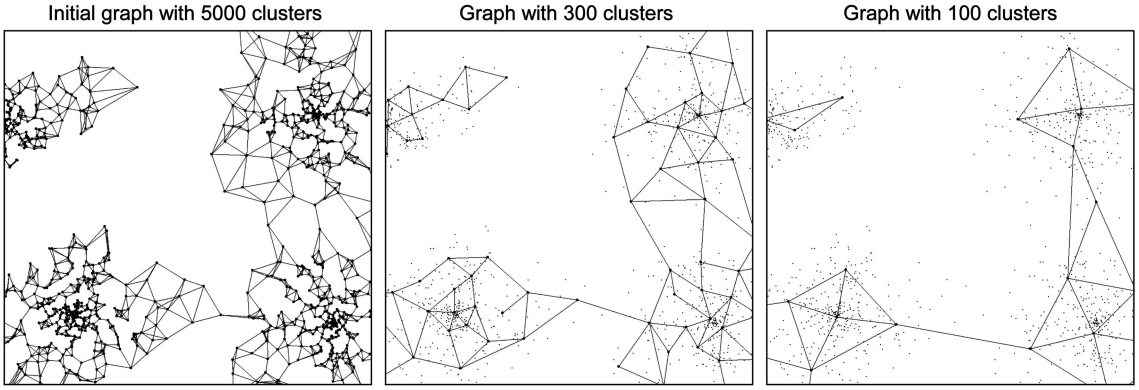


Fig. 3. An example of the graph with a synthetic data set and the corresponding k-NN graph ($k = 4$). Darker dots illustrate cluster centroids. Arrowheads are not printed for clarity.

TABLE 1
The Estimated Number of Steps and Distance Calculations per Iteration

	Fast exact PNN		Proposed (simple)		Proposed (double-linked)	
	Steps	Distances	Steps	Distances	Steps	Distances
Find the best pair	N	-	1	-	1	-
Merge	N	-	$k^2 + \log N$	k	$k^2 + \tau k + \log N$	k
Remove obsolete	N	-	$k + \log N$	-	$\log N$	-
Find neighbors	N	-	kN	-	τk	-
Update distances	$N(1+\tau)$	τN	$\tau + \tau/k \cdot \log N$	τ	$\tau + \tau/k \cdot \log N$	τ
Total	$O(\tau N^2)$	$O(\tau N^2)$	$O(kN^2)$	$O(\tau N)$	$O(\tau N \cdot \log N)$	$O(\tau N)$

way to find the neighbors is to browse through the k-NN lists of all N clusters. One of the merged clusters (cluster b in our case) must also be removed (*Remove obsolete*), which takes $O(k + \log N)$ time due to the removal of k pointers and due to the removal of one value from the heap.

Finally, we recalculate the distance values of the neighbor clusters (*Update distances*). This takes $O(\tau + (\tau/k) \cdot \log N)$ time, where τ is the in-degree of the node. There are τ distance values to be updated, each taking only $O(1)$. However, only the first entry from each k-NN list is stored in the heap. We, therefore, approximate that only every kth of the distance update generates heap update. Thus, there are τ/k heap updates on average, each requiring $O(\log N)$ time.

The number of τ depends both on k and on the data. Every cluster has exactly k outgoing pointers and the in-degree of a randomly chosen node must therefore also be k, on average. As there are two clusters involved in the merge, $\tau = 2k$ for a randomly chosen cluster pair. However, the pair to be merged is more likely to be selected from a dense area. Thus, the number of neighbors depends also on the dimensionality and structure of the data, but in a way that is not trivial to measure.It has been shown in [18] that, in Euclidean space, τ is upper bounded by k times the *kissing number*, which is defined as the number of unit hyperspheres that are touching another unit hypersphere without any intersections. The kissing numbers are known for some dimensions, but, in general, the problem is unsolved [19]. It was observed in [3] that there are only a relatively few (about 2-4) neighbors, on average.

3.3 Double Linked Algorithm (DLA)

Even though the number of distance calculations has been greatly reduced by the simple algorithm, the number of steps still sums up to $O(kN^2)$; see Table 1. The bottleneck is the search for the incoming edges, i.e., the nodes that consider the merged cluster as their neighbor. This step dominates the time complexity requiring $O(kN)$ per iteration, thus summing up to $O(kN^2)$ in total.

To attack this problem, we implement a double linked list structure as shown in Fig. 4. For every node, we maintain two lists: the k-NN list containing the pointers to the k nearest neighbors and another containing the "back pointers" to the clusters that consider the particular cluster as part of their k nearest neighbors. For example, in Fig. 4, there are seven clusters that consider either a or b as their nearest neighbors. All of them appear in the back pointer lists of a (h, j, k, b) or b (c, d, a). The differences between the simple algorithm and the double link algorithm (DLA) are described next.

In the merge step, in addition to resolving the k-NN list, we must also update the back pointers of the neighbors. This is done by browsing through the back pointer lists of the neighbors. As there are k neighbors to be checked (clusters c and a in Fig. 4) and the number of back pointers is τ, an additional $O(\tau k)$ term appears in the time complexity of this step. On the other hand, the back pointer lists eliminate the need for the $O(kN)$ time loops in the search of the neighbors and replace it by searching through the back pointer list of the cluster, which takes $O(\tau k)$ time. In Fig. 4, we need to consider the neighbors h, j, k, c, and d. The rest of the algorithm is the same and also the number of distance calculations remains the same.

In the double linked algorithm, there is no clear bottleneck anymore. If we consider k as a constant, we can simplify the overall time complexity to $O(\tau N \cdot \log N)$ originating from the heap updates ($\tau/k \cdot \log N$); see Table 1. In practice, τ is also very small for realistic data sets and, therefore, the merge step takes more

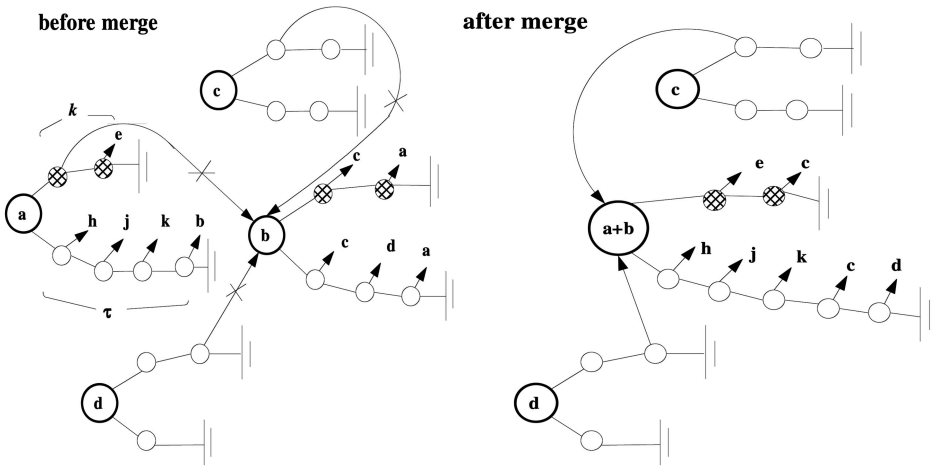


Fig. 4. Illustration of the double linked list structure for a part of the graph presented in Fig. 2 and the required updates due to the merge of clusters a and b.

TABLE 2
The Observed Cumulative Number of Steps and Distance Calculations ($k = 5$) for *Bridge*

	Fast exact PNN		Proposed (simple)		Proposed (double-linked)	
	Steps	Distances	Steps	Distances	Steps	Distances
Find the best pair	8 357 760	-	3 840	-	3 840	-
Merge	8 367 905	-	200 144	22 613	305 326	22 493
Remove obsolete	8 349 185	-	102 450	-	45 514	-
Find neighbors	8 357 760	-	41 769 600	-	204 689	-
Update distances	48 538 136	40 166 328	198 334	57 800	187 123	46 566
Total	81 970 746	40 166 328	42 274 368	80 413	746 492	69 059

computation because $\tau k \geq \tau/k \cdot \log N \Leftrightarrow k^2 \geq \log N$, which is true for our data sets ($k = 5$ and $N = 1{,}000..100{,}000$).

The observed number of steps and distance calculations are reported in Table 2 for a sample data set (see Section 5). The number of distance calculations is reduced to a fraction of that of the full search (from 40 million to 80,413) by the simple implementation, but the number of steps only to 52 percent of that of the full search (from 81 million to 42 million). The double link algorithm, however, reduces the number of steps to 746,492.

4 Creation of the Neighborhood Graph

The exact k-NN graph can be constructed in $\mathrm{O}(N^2)$ time using *brute force* by considering all pairwise distances. We therefore consider faster alternatives for constructing an approximate k-NN graph by using *K-d tree*, *divide-and-conquer*, and *projection-based search*.

The K-d tree method has been widely used for finding the nearest neighbor [9], [20] and, here, we extended it for finding k nearest neighbors. The method creates a tree-structured partition by recursively splitting the data into two disjoint partitions along one of the major axes. The leaf nodes of the resulting tree form disjoint partitions called *buckets*. The nearest neighbor for any vector can be found among the vectors in the same bucket, or from its sibling nodes. The creation of the tree takes $\mathrm{O}(N \cdot \log N)$ time and each search $\mathrm{O}(\log N)$ time, but only if the dimensionality d is considered as a constant.

The *closest pair problem* is stated as follows: Given N points in d-dimensional space, find the two whose mutual distance is the smallest. The problem can be solved by the divide-and-conquer approach [10] that recursively divides the data set into subsets. We first calculate the principal axis of the data vectors and select a $(d-1)$-dimensional hyperplane perpendicular to the axis, so that it divides the set approximately into two halves. A third subset is generated from vectors that are closer to the dividing *hyperplane H* than its nearest neighbor. The three subproblems are solved recursively and the results are then combined. An upper bound for the time complexity can be estimated by the recurrence

$$T(N) = 3 \cdot T(N/2) + \mathrm{O}(N \cdot d^2),$$

which derives to $\mathrm{O}(d^2 \cdot N^{1.58})$. Lower time complexity could be obtained by dividing the hyperplane by a simpler heuristic or by making tighter bounds for the size of the third subset.

Mean-distance ordered partial search (MPS) calculates projections of the data vectors to the diagonal axis [11] and proceeds to search bidirectionally along the axis, starting from the input vector. Given the distance to the best candidate found so far, other vectors can be excluded from the search if the projection is outside of a given radius. The precondition can be calculated in $\mathrm{O}(1)$ time and, if it holds, the $\mathrm{O}(d)$ time distance calculation can be avoided.

For finding the k nearest vectors, we relax the condition of the graph and find any k neighbors, instead of the nearest ones. We use the MPS method for finding the nearest neighbor, but stop the search when it has been found (*MPS full*) or when a predefined search limit has been reached (*MPS limited*). In addition to this, we maintain an ordered list of the k best candidates found so far. The rest of the neighbors are taken as the k best candidates from the list of the candidates once the first nearest neighbor has been found. It

TABLE 3
Summary of the Data Sets

Data set	Type of data set	Number of data vectors (N)	Number of clusters (M)	Dimension of data vector (d)
Bridge	Gray-scale image	4086	256	16
House	*RGB* image	34112	256	3
Miss America	Residual vectors	6480	256	16
$BIRCH_1$-$BIRCH_3$	Synthetically generated	100000	100	2
*Dim*64-1024	Synthetically generated	1000	256	64 – 1024

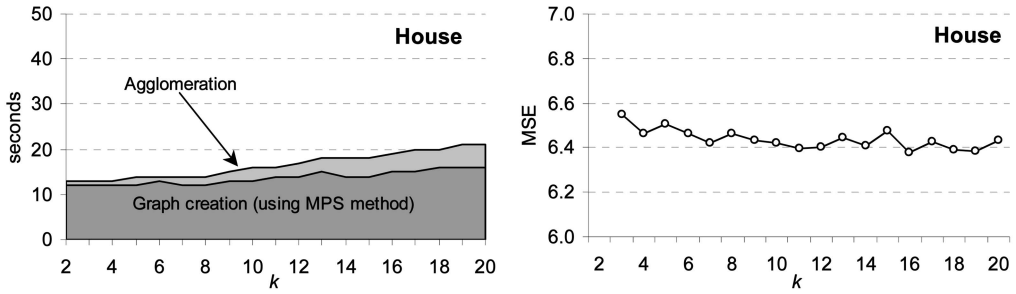


Fig. 5. The running time and the quality of the *DLA* as functions of k.

is expected that the rest of the candidates are nearby vectors, although not necessarily the nearest ones.

The main advantage of the method is its simplicity and the main disadvantage is that the worst case time complexity is still $O(N^2)$. For details of the algorithm, see [21]. A better projection axis can be obtained by *principal component analysis* (PCA) [22, p. 10] on the data set and by using the projection to the first principal component. It is expected that the search can be terminated earlier by using the principal axis rather than the diagonal one. However, the calculation of the principal axis takes $O(Nd^2)$ time, which might outweigh the additional speedup in the case of higher dimensional data sets.

TABLE 4
Observed Number of Nearest Neighbor Updates (τ) Required

Data set	τ
Bridge	12.1
House	7.0
Miss America	11.1
$BIRCH_1$	5.1
$BIRCH_2$	5.0
$BIRCH_3$	5.1

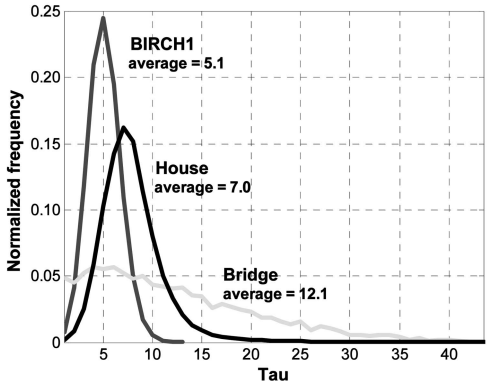


Fig. 6. Histograms of the observed τ values.

5 Experiments

We consider three image data sets from [3], three *BIRCH* data sets [23], and five high dimensional data sets *Dim64* to *Dim1024* with higher dimensionality varying from 64 to 1,024. The summary of

TABLE 5
Running Times of the Graph Creation Algorithms for the Image Data Sets ($k = 5$)

Algorithm:	*Bridge*		*House*		*Miss America*	
	Time	MSE	Time	MSE	Time	MSE
Brute force	34	170.25	881	6.28	89	5.40
KD-tree	12	169.83	5	6.33	79	5.40
D-n-C	2	171.58	49	6.37	7	5.44
MPS full	3	170.28	19	6.34	45	5.41
MPS limited	3	170.60	14	6.43	6	5.59
MPS / PCA	10	170.23	37	6.28	58	5.38

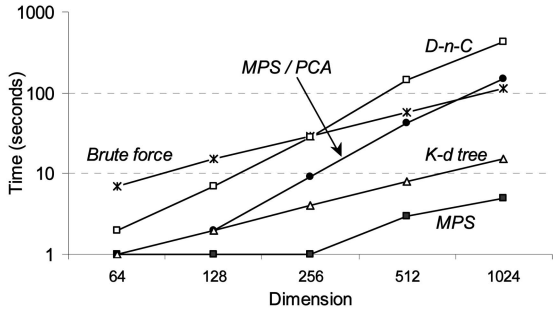


Fig. 7. Running times of the graph creation algorithms versus dimensionality (*Dim* data sets).

TABLE 6
Time (Seconds) and Quality (MSE) Comparison of the Methods

Image data sets		*Bridge*		*House*		*Miss America*	
		Time	MSE	Time	MSE	Time	MSE
Original PNN	[12]	>9999	168.92	>9999	6.27	>9999	5.36
K-d tree PNN	[12]	2	199.20	65	7.03	5	6.31
Fast exact PNN	[3]	79	168.92	1574	6.27	229	5.36
+PDS+MPS+Lazy	[15]	9	168.92	190	6.26	106	5.37
DLA	Proposed	3	170.60	14	6.43	6	5.59
K-means	[24]	13	179.87	23	7.81	20	5.96
+PDS+MPS+Activity	[25]	2	179.87	3	7.81	8	5.96
PNN + k-means	Combined	80	165.04	191	6.07	109	5.24
DLA + k-means		4	166.35	15	6.13	9	5.34

Birch data sets		$BIRCH_1$		$BIRCH_2$		$BIRCH_3$	
		Time	MSE	Time	MSE	Time	MSE
Original PNN	[12]	>9999	4.73	>9999	2.28	>9999	1.96
K-d tree PNN	[12]	401	7.09	401	3.86	401	2.41
Fast exact PNN	[3]	>9999	4.73	>9999	2.28	>9999	1.96
+PDS+MPS+Lazy	[15]	2397	4.73	2115	2.28	2316	1.96
DLA	Proposed	40	4.74	16	2.28	34	2.01
K-means	[24]	209	5.52	43	7.99	171	2.53
+PDS+MPS+Activity	[25]	29	5.52	8	7.99	35	2.53
PNN + k-means	Combined	2401	4.64	2116	2.28	2333	1.88
DLA + k-means		44	4.64	17	2.28	51	1.91

the data sets is presented in Table 3. The results are run on a 450 MHz Pentium III personal computer.

In all test sets, a relatively small neighborhood size (k) is sufficient to keep the distortion close to that of the full search. The actual neighborhood size is not so critical, as long as the neighborhood is large enough to keep the vectors within the same cluster connected, thus avoiding isolated subclusters. According to our experiments, this can be reached in all test sets presented here by using $k = 3$ or $k = 4$.

The running time has linear dependency with the parameter k, but the growing rate is small; see Fig. 5. The graph creation is the bottleneck of the algorithm. In the rest of the paper, we fix the neighborhood size at $k = 5$. The number of incoming pointers (τ) depends on the data set and on the value of k. The observed numbers for the data sets have been reported in Table 4, and their distributions in Fig. 6. The results clearly show that the number of τ increases with the dimensionality of the data set.

For the graph creation, we consider the following five algorithms:

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

The *D-n-C* algorithm works well with some of the image data sets (see Table 5), but, as the results in Fig. 7 demonstrate, increasing dimensionality affects its performance severely because of the calculation of the *PCA*, which takes quadratic time. *K-d tree* works best for *House* because of its low dimensionality. For higher dimensional data sets, *MPS* provides the best overall results (limited to 500 searches) and it is, therefore, chosen for the rest of the experiments.

Finally, the proposed method is compared against the existing *PNN* variants and the *k-means* algorithm in Table 6. The *original PNN* includes both the slow $O(N^3)$ and the faster $O(N \cdot \log N)$ time inexact variant using *K-d tree*. The *Fast exact PNN* has two variants: the one proposed in [3] and an improved variant [15] that uses *PDS*, *MPS*, and *Lazy evaluation* for speedup. The *k-means* results include the original method [24] and a faster variant that uses *PDS*, *MPS* and *activity detection* for speedup [25]. The results of the combined methods (*PNN + k-means*, *DLA + k-means*) are produced by feeding the output of the *PNN* (or *DLA*) as the input to the *k-means*.

The proposed method (*DLA*) provides clustering almost as good as (*Birch* sets) or only slightly worse (image sets) than the *PNN*, but significantly faster. The difference is remarkable, especially with the large data sets (*Birch*). Comparison to *k-means* shows that the proposed method provides significantly better

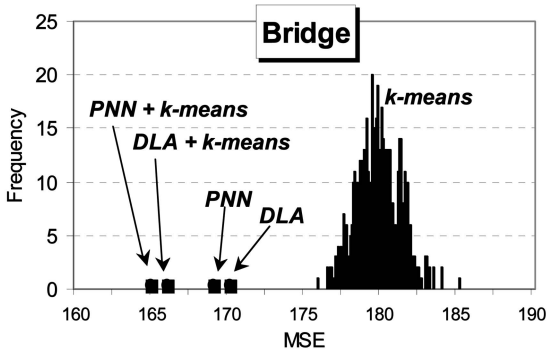
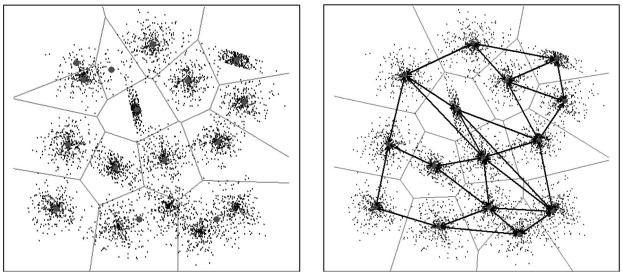


Fig. 8. Histograms of the MSE-values of 500 runs of the *k means*, *PNN*, and *DLA*.



(a) (b)

Fig. 9. Sample clustering of a synthetic data set (a) by *k-means* and (b) by DLA with the remaining k-NN graph.

quality with an algorithm that is competitive also in speed. The difference in quality is demonstrated in Fig. 8 and Fig. 9.

6 Conclusions

A fast agglomerative method has been proposed for clustering using an approximate k-nearest neighbor graph. A relatively small neighborhood size is sufficient to produce clustering with similar quality to that of the full search. There are no theoretical grounds for how to fix the exact neighborhood size optimally, but there is one guideline that should be followed: The connectivity of the vectors within the clusters should be preserved. The bottleneck of the algorithm is the graph creation and it remains a challenge to invent a practical algorithm for creating a reasonably accurate k-NN graph in subquadratic time.

References

[1] J.H. Ward, "Hierarchical Grouping to Optimize an Objective Function," *J. Am. Statisical Assoc.*, vol. 58, pp. 236-244, 1963.

[2] J. Shanbehzadeh and P.O. Ogunbona, "On the Computational Complexity of the LBG and PNN Algorithms," *IEEE Trans. Image Processing*, vol. 6, no. 4, pp. 614-616, Apr. 1997.

[3] P. Fränti, T. Kaukoranta, D.-F. Shen, and K.-S. Chang, "Fast and Memory Efficient Implementation of the Exact PNN," *IEEE Trans. Image Processing*, vol. 9, no. 5, pp. 773-777, May 2000.

[4] J.C. Gover and G.J.S. Ross, "Minimum Spanning Trees and Single Linkage Cluster Analysis," *Applied Statistics*, vol. 18, pp. 54-64, 1969.

[5] S. Bandyopadhyay, "An Automatic Shape Independent Clustering Technique," *Pattern Recognition*, vol. 37, no. 1, pp. 33-45, Jan. 2004.

[6] P.H.A. Sneath, "The Application of Computers to Taxonomy," *J. General Microbiology*, vol. 17, no. 1, pp. 210-226, Aug. 1957.

[7] G. Karypis, E. Han, and V. Kumar, "CHAMELEON: A Hierarchical Clustering Algorithm Using Dynamic Modeling," *Computer*, vol. 32, no. 8, pp. 66-75, Aug. 1999.

[8] D. Harel and Y. Koren, "Clustering Spatial Data Using Random Walks," *Proc. Seventh ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '01)*, pp. 281-286, Aug. 2001.

[9] J.H. Friedman, J.L. Bentley, and R.A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Trans. Math. Software*, vol. 3, no. 3, pp. 209-226, Sept. 1977.

[10] O. Virmajoki and P. Fränti, "Divide-and-Conquer Algorithm for Creating Neighborhood Graph for Clustering," *Proc. Int'l Conf. Pattern Recognition (ICPR '04)*, vol. 1, pp. 264-267, Aug. 2004.

[11] S.-W. Ra and J.K. Kim, "A Fast Mean-Distance-Ordered Partial Codebook Search Algorithm for Image Vector Quantization," *IEEE Trans. Circuits and Systems*, vol. 40, no. 9, pp. 576-579, Sept. 1993.

[12] W.H. Equitz, "A New Vector Quantization Clustering Algorithm," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 37, no. 10, pp. 1568-1575, Oct. 1989.

[13] T. Kurita, "An Efficient Agglomerative Clustering Algorithm Using a Heap," *Pattern Recognition*, vol. 24, no. 3, pp. 205-209, Mar. 1991.

[14] T. Kaukoranta, P. Fränti, and O. Nevalainen, "Vector Quantization by Lazy Pairwise Nearest Neighbor Method," *Optical Eng.*, vol. 38, no. 11, pp. 1862-1868, Nov. 1999.

[15] O. Virmajoki, P. Fränti, and T. Kaukoranta, "Practical Methods for Speeding-Up the Pairwise Nearest Neighbor Method," *Optical Eng.*, vol. 40, no. 11, pp. 2495-2504, Nov. 2001.

[16] O. Virmajoki and P. Fränti, "Fast Pairwise Nearest Neighbor Based Algorithm for Multilevel Thresholding," *J. Electronic Imaging*, vol. 12, no. 4, pp. 648-659, Oct. 2003.

[17] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*. Cambridge, Mass.: MIT Press, 1998.

[18] G.L. Miller, S-H. Teng, W. Thurston, and S.A. Vavaris, "Separators for Sphere-Packings and Nearest Neighbor Graphs," *J. ACM*, vol. 44. no. 1, pp. 1-29, Jan. 1997.

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

[20] R. Sproull, "Refinements to Nearest-Neighbor Searching in k-d Trees," *Algorithmica*, vol. 6, pp. 579-589, 1991.

[21] P. Fränti, O. Virmajoki, and V. Hautamäki, "Fast PNN-Based Clustering Using *k* Nearest Neighbor Graph," *Proc. IEEE Int'l Conf. Data Mining (ICDM '03)*, pp. 525-528, Nov. 2003.

[22] *Encyclopedia of Statistical Sciences*, vol. 6, S. Kotz, N.L. Johnson, and C.B. Read, eds. New York: John Wiley Sons, 1985.

[23] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: A New Data Clustering Algorithm and Its Applications," *Data Mining and Knowledge Discovery*, vol. 1, no. 2, pp. 141-182, June 1997.

[24] Y. Linde, A. Buzo, and R.M. Gray, "An Algorithm for Vector Quantizer Design," *IEEE Trans. Comm.*, vol. 28, no. 1, pp. 84-95, Jan. 1980.

[25] T. Kaukoranta, P. Fränti, and O. Nevalainen, "A Fast Exact GLA Based on Code Vector Activity Detection," *IEEE Trans. Image Processing*, vol. 9, no. 8, pp. 1337-1342, Aug. 2000.