Fast and Versatile Algorithm for Nearest Neighbor Search Based on a Lower Bound Tree

Yong-Sheng Chen [a,*], Yi-Ping Hung [b,c], Ting-Fang Yen [a], Chiou-Shann Fuh [b]

[a] *Department of Computer Science, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu 300, Taiwan*

[b] *Department of Computer Science and Information Engineering, National Taiwan University, 1 Roosevelt Road, Section 4, Taipei 106, Taiwan*

[c] *Institute of Information Science, Academia Sinica, 128 Academia Road, Section 2, Taipei 115, Taiwan*

Abstract

In this paper, we present a fast and versatile algorithm which can rapidly perform a variety of nearest neighbor searches. Efficiency improvement is achieved by utilizing the distance lower bound to avoid the calculation of the distance itself if the lower bound is already larger than the global minimum distance. At the preprocessing stage, the proposed algorithm constructs a lower bound tree (LB-tree) by agglomeratively clustering all the sample points to be searched. Given a query point, the lower bound of its distance to each sample point can be calculated by using the internal node of the LB-tree. To reduce the amount of lower bounds actually calculated, the winner-update search strategy is used for traversing the tree. For further efficiency improvement, data transformation can be applied to the sample and the query points. In addition to finding the nearest neighbor, the proposed algorithm can also (i) provide the k-nearest neighbors progressively; (ii) find the nearest neighbors within a specified distance threshold; and (iii) identify neighbors whose distances to the query are sufficiently close to the minimum distance of the nearest neighbor. Our experiments have shown that the proposed algorithm can save substantial computation, particularly when the distance of the query point to its nearest neighbor is relatively small compared with its distance to most other samples (which is the case for many object recognition problems).

Key words: Nearest neighbor search; Lower bound tree

* Corresponding author. Tel.: +886-3-5131316; fax: +886-3-5724176.
Email addresses: yschen@cs.nctu.edu.tw (Yong-Sheng Chen), hung@csie.ntu.edu.tw (Yi-Ping Hung), tyen@andrew.cmu.edu (Ting-Fang Yen), fuh@csie.ntu.edu.tw (Chiou-Shann Fuh).

1 Introduction

Nearest neighbor search has been widely applied in many fields, including object recognition [1], pattern classification and clustering [2,3], image matching [4,5], data compression [6,7], texture synthesis [8], and information retrieval in database systems [9,10]. Depending on the application, each object (pattern, image block, or other kind of data) can be represented as a multi-dimensional point. Using a distance function as the measure of dissimilarity, the nearest neighbor search for the most similar object can be regarded as the closest point search in a multi-dimensional space. In general, a fixed data set P of s sample points in a d-dimensional space is given, represented by $P = \{\mathbf{p}_i \in R^d | i = 1, \ldots, s\}$. Preprocessing can be performed, if necessary, to construct a particular data structure. The goal of the nearest neighbor search is to find in P the point closest to each query point $\mathbf{q}$ in the d-dimensional space. A straightforward way to do so is to exhaustively compute and compare the distances between the query point and all sample points. This exhaustive search has the computational complexity of $O(s \cdot d)$, and when one or both of s, d are large, the process can be very time-consuming.

Many methods have been proposed to speed up the computation of nearest neighbor search. One category of these methods partitions the data space into "regions" according to the sample points. Various shapes of region have been adopted, including hyper-rectangular bucket (the k-d tree method [11]), bounding rectangle (the R-tree [12] and the SR-tree [13] methods), bounding sphere (the SS-tree [14] and the SR-tree [13] methods), pyramid [15], and Voronoi cell [16]. A data structure, usually a tree, was used for recording and indexing these regions. Given a query point, its nearest neighbor can be found by using the tree structure. For example, Bentley (the k-d tree method [11]) partitioned the data space into hyper-rectangular buckets, each of which contains several sample points. Their method for nearest neighbor search is performed by a binary search for the target bucket followed by a local search for the desired sample point in the target bucket and its neighboring buckets, which is very efficient when the dimension of the data space is small. However, as reported in [16] and [17], when the number of dimensions increases, its performance degrades exponentially, in an effect known as the *curse of dimensionality*. The main reason for this phenomenon is that more neighboring buckets must be checked when the dimension is higher. Thus, the number of sample points to be examined increases dramatically.

Another category of fast nearest neighbor search methods are elimination-based methods (see [18] for a review). For example, Fukunaga and Narendra [19] constructed a tree structure for the sample points, and used a branch-and-bound search strategy to traverse and prune the tree structure in the query process for efficiently determining the nearest neighbor. To construct

the tree structure, a set of sample points is first divided into k subsets using the k-means clustering algorithm. Each subset is then further divided into k subsets. This process is repeated thereby creating a tree structure, with each node in the tree representing a number of sample points. The mean of these sample points and the farthest distance from the mean to these sample points are recorded. For a node in the tree, if the distance between its recorded mean and the query point subtracted by the recorded farthest distance is larger than the minimum distance obtained so far, the distance computation for all the sample points represented by this node can be avoided due to the triangle inequality. Brin [20] proposed a method similar to [19]. They constructed another kind of data structure, called GNAT, by using hierarchical decomposition for the sample points. Each level in the GNAT data structure can have different numbers of branches. Vidal [21] also utilized the branch-and-bound search strategy to reduce the distance calculations. Friedman et al. [22] proposed a projection based search algorithm. On the projection coordinate, the sample points are sorted according to the values of this coordinate. They are then examined in the order of their distance (on this coordinate) to the query point. Sample points whose distance to the query point on the projection coordinate is larger than the current minimum distance (on all coordinates) can be eliminated, thereby speeding up the search process. Soleymani and Morgera [23] used an elimination technique similar to [22] where they performed the elimination test on each coordinate, instead of only on the projection coordinate. Djouadi and Bouktache [24] partitioned the underlying space of the sample points into a set of cells. By calculating the distances between the query point and the centers of the cells, the nearest neighbor can be found efficiently by searching only in those cells in th vicinity of the query point, rather than the whole space. Lee and Chae [25] also proposed a fast nearest neighbor search method, which uses a number of anchor sample points to eliminate many distance calculations based on the triangle inequality. In [26], McNames presented a fast nearest-neighbor algorithm based on principal axis trees. This method utilizes depth-first search and distance lower bounds to eliminate many distance calculations.

Instead of finding the *exact* nearest neighbor, that is, the global optimum, another research direction is to find the *approximate* nearest neighbor. Arya et al. [27] proposed a fast algorithm to find the $(1 + r)$-approximate nearest neighbor within a factor of $(1+r)$ of the distance between the query point and its exact nearest neighbor. They constructed a balanced box-decomposition (BBD) tree by hierarchically decomposing the underlying space. A priority search is then applied to efficiently find the approximate nearest neighbors.

There are some application-dependent issues worth considering for nearest neighbor search. For example, Faragó et al. [28] presented a fast nearest-neighbor search algorithm in dissimilarity space in which the triangle inequality may not hold [29]. In database systems, the obtained query results may

have to be checked using some other conditions in addition to the minimum distance requirement. In this case, the number k of the k-nearest neighbors cannot be specified beforehand. Hjaltason and Samet [30] proposed a fast algorithm which can provide k-nearest neighbors progressively (one-by-one) until the required number of nearest neighbors satisfying other conditions is obtained. In object recognition applications, the nearest neighbor of a query object is of interest only when the distance between the query object and its nearest neighbor is small enough. For this kind of applications, Nene and Nayar [17] proposed a fast algorithm for searching the nearest neighbor within a pre-specified small distance threshold in a high-dimensional space. For each dimension, their method excludes the sample points whose distances to the query point at the current dimension are larger than the distance threshold. The nearest neighbor can then be determined from examining the remaining candidates. This process may eliminate all the sample points if the distance threshold is too small. The remedy is to enlarge the distance threshold gradually. For some other applications that utilize audio or image matching, each multi-dimensional sample or query point represents an autocorrelated signal. That is, the signal values in consecutive dimensions are correlated. In such cases, the search process can be accelerated by applying some data transformation to each data point, such as mean pyramid construction [6,31,5] or wavelet transform [7].

In this paper, a novel algorithm is presented which efficiently searches for the exact nearest neighbor in Euclidean space. The proposed algorithm first prepocesses the sample points by constructing a *lower bound tree* (*LB-tree*), in which each leaf node represents a sample point and each internal node represents a mean point in a space of smaller dimension. For each query point, a lower bound of its distance to each sample point can be calculated by using a mean point of an internal node in the LB-tree. Distance calculations can be avoided for many sample points whose lower bound of the distance to the query point is larger than the minimum distance between the query point and its nearest neighbor. The whole search process is accelerated this way because the computational cost of the lower bounds is less than that of the distance. In addition to the use of an LB-tree, the following three techniques are further adopted to reduce lower-bound calculation:

1. **Winner-update search** To reduce the number of nodes examined, we apply a winner-update search strategy for traversing the LB-tree. Starting from the root node of the LB-tree, the node having minimum lower bound is replaced by its children for the following competition after the lower bounds of these children having been calculated.
2. **Agglomerative clustering** When constructing the LB-tree, we use an agglomerative clustering technique to keep the number of the internal nodes as small as possible while keeping the lower bound as tight as possible.
3. **Data transformation** Data transformation, such as the wavelet trans-

form or the principal component analysis, is applied to each point so that the lower bound of an internal node can be further tightened, thus saving more computation.

Among the above three techniques, both the winner-update search strategy and the data transformation are performed in the search process. That means, additional computations are required for each query. Fortunately, the amount of this increased burden is relatively small compared to the savings gained by these two techniques and the overall search efficiency can be improved in most situations (see Section 7). The other technique, agglomerative clustering for LB-tree construction, can be very time-consuming. However, the LB-tree is constructed in the preprocessing stage. It is usually worthwhile to obtain a good data structure at the expense of large amount of computation beforehand for the sake of high search efficiency. For example, it took about three hours to construct the LB-tree for $36,000$ sample points in 35-dimensional space and the search process can be over one thousand times faster by using the constructed LB-tree in our experiment (see Section 7.3).

Our experiments have shown that the proposed algorithm for nearest neighbor search can save a considerable amount of computation, particularly when the query point is relatively closer to its nearest neighbor than to most other samples. Furthermore, the proposed algorithm is versatile because it can deal with various types of queries. More specifically, this algorithm can speed up the progressive search for k-nearest neighbors, the search for nearest neighbors within a specified distance threshold, and the search for neighbors whose distances to the query are sufficiently close to the minimum distance of the nearest neighbor.

This paper is organized as follows. At first, we introduce the data structure and the proposed algorithm for nearest neighbor search in Sections 2 and 3, respectively. Next, we present the supplement of the proposed algorithm for other query types in Section 4. Then, the construction of the LB-tree is described in Section 5. Two kinds of data transformation are introduced in Section 6. Section 7 presents the experimental results of the proposed algorithm. Finally, conclusions are stated in Section 8.

2 Multilevel Structure and LB-Tree

This section introduces the LB-tree used in the proposed algorithm for nearest neighbor search. We will first describe the multilevel structure of a data point. The multilevel structures of all the sample points can then be used to construct the LB-tree. We shall also introduce some properties of the LB-tree, which reveals the effectiveness of the proposed algorithm.

2.1 Multilevel Structure of Each Data Point

For a point $\mathbf{p} = [p_1, p_2, \ldots, p_d]$ in a d-dimensional Euclidean space, R^d, we denote its multilevel structure of $L+1$ levels by $\{\mathbf{p}^0, \mathbf{p}^1, \ldots, \mathbf{p}^L\}$, and define it in the following. At each level l, $\mathbf{p}^l = [p_1, p_2, \ldots, p_{d^l}]$, which comprises the first d^l dimensions of the point $\mathbf{p}$, is referred to as the level-l projection of $\mathbf{p}$, for $1 \leq d^l \leq d$, $l = 0, \ldots, L$. A trivial way to construct a d-level structure is to let $d^l = l + 1$, $l = 0, \ldots, d-1$. Here, d^l is an increasing function of l because $d^l = l + 1 < (l+1) + 1 = d^{l+1}$. Notice that the construction method of multilevel structure for a data point belongs to one kind of telescoping functions, which can be used to contract and extend feature vectors, proposed by Lin et al. [32].

In this paper the dimension at level l is set to $d^l = 2^l$. Without loss of generality, we assume that the dimension of the data point space, d, is equal to 2^L. If d is not a power of 2, zero padding can be used to enlarge the dimension of the underlying space. In this way, an $(L+1)$-level structure for point $\mathbf{p}$ can be constructed. Notice that level-L projection, $\mathbf{p}^L$, is equivalent to point $\mathbf{p}$. An example of a 4-level structure, $\{\mathbf{p}^0, \ldots, \mathbf{p}^3\}$, where $d = 8$, is shown in Figure 1.

Given the multilevel structures of points $\mathbf{p}$ and $\mathbf{q}$, we can derive the succeeding inequality property:

Property 1 *The Euclidean distance between* $\mathbf{p}$ *and* $\mathbf{q}$ *is larger than or equal to the Euclidean distance between their level-l projections* $\mathbf{p}^l$ *and* $\mathbf{q}^l$ *for each level l. That is,*

$$\|\mathbf{p} - \mathbf{q}\|_2 \geq \|\mathbf{p}^l - \mathbf{q}^l\|_2, \ l = 0, \ldots, L.$$

Although all the properties shown in this section (and hence the proposed algorithm) are valid for any l_p norm, we adopt l_2 norm (Euclidean distance) here. The reason is that if the data transformation described in Section 6 is applied, the distance other than l_2 norm may change. From Property 1, a lower bound of the distance $\|\mathbf{p} - \mathbf{q}\|_2$ can be considered to be the distance $\|\mathbf{p}^l - \mathbf{q}^l\|_2$ calculated using the level-l projections. Notice that the computational complexity of the distance $\|\mathbf{p}^l - \mathbf{q}^l\|_2$ is less than that of the distance $\|\mathbf{p} - \mathbf{q}\|_2$. Specifically, the complexity of calculating the distance between level-l projections is $O(2^l)$ for $l = 0, \ldots, L$.

This section introduces the LB-tree and some of its properties. To construct an LB-tree, we require the multilevel structures of all sample points $\mathbf{p}_i$, $i = 1, \ldots, s$, in a data set P, where s is the number of the data points in P. The LB-tree has the same number of levels as the multilevel structure, without considering the dummy root node, which is considered to have zero dimension. At level L in the LB-tree, each leaf node contains a level-L projection ${\mathbf{p}_i}^L$, which is equivalent to the sample point $\mathbf{p}_i$. For level 0 to level $L-1$, the level-l projections, ${\mathbf{p}_i}^l$, $i = 1, \ldots, s$, of all the sample points can be clustered to form a hierarchy, as illustrated in Figure 2, where $L = 3$ and $s = 9$. More details of the LB-tree construction will be given in Section 5.

Let s^l denote the number of clusters at level l. Let $\langle \mathbf{p} \rangle$ denote the node containing the point $\mathbf{p}$ in the LB-tree. Each cluster ${C^l}_j$, $j = 1, \ldots, s^l$, is represented by an internal node $\langle {\mathbf{m}^l}_j \rangle$ at level l in the LB-tree. The internal node $\langle {\mathbf{m}^l}_j \rangle$ contains the mean point ${\mathbf{m}^l}_j$, which is the mean of all the level-l projections of the sample points contained in this cluster, and the associated *radius*, ${r^l}_j$, which is the radius of the smallest 2^l-dimensional hyper-sphere centered at ${\mathbf{m}^l}_j$ and covers all the level-l projections in cluster ${C^l}_j$. An example of an LB-tree is shown in Figure 3. This smallest hyper-sphere is called the *bounding sphere* of ${C^l}_j$; its radius can be calculated as the maximum distance from the mean point ${\mathbf{m}^l}_j$ to all level-l projections in this cluster. The LB-tree has the following inequality property:

Property 2 *Given a sample point $\mathbf{p}^*$, the distance between its level-l projection, $\mathbf{p}^{*l}$, and its level-l ancestor, ${\mathbf{m}^l}_{j^*}$, is smaller than or equal to the radius of the bounding sphere of cluster ${C^l}_{j^*}$. That is,*

$$\|\mathbf{p}^{*l} - {\mathbf{m}^l}_{j^*}\|_2 \leq {r^l}_{j^*}, \;\; l = 0, \ldots, L.$$

Notice that a leaf node is equivalent to a cluster of only one point. The radius is zero and the mean point is the sample point itself in such cases.

Now, given a query point $\mathbf{q}$, we first construct its multilevel structure as described in Section 2.1. For a sample point $\mathbf{p}^*$ and its corresponding leaf node $\langle \mathbf{p}^* \rangle$, its ancestor at level l in the LB-tree is denoted $\langle {\mathbf{m}^l}_{j^*} \rangle$. As illustrated in Figure 4, the following inequality can be derived using the triangle inequality and Properties 1 and 2:

$$\begin{aligned}
\|\mathbf{p}^* - \mathbf{q}\|_2 &\geq \|\mathbf{p}^{*l} - \mathbf{q}^l\|_2 \\
&\geq \|{\mathbf{m}^l}_{j^*} - \mathbf{q}^l\|_2 - \|\mathbf{p}^{*l} - {\mathbf{m}^l}_{j^*}\|_2 \\
&\geq \|{\mathbf{m}^l}_{j^*} - \mathbf{q}^l\|_2 - {r^l}_{j^*}. \qquad (1)
\end{aligned}$$

The LB-distance $d_{LB}(\langle \mathbf{m}^l{}_{j^*}\rangle, \mathbf{q}^l)$ between the internal node $\langle \mathbf{m}^l{}_{j^*}\rangle$ and $\mathbf{q}^l$, the level-l projection of the query point $\mathbf{q}$, is defined as:

$$d_{LB}(\langle \mathbf{m}^l{}_{j^*}\rangle, \mathbf{q}^l) \equiv \|\mathbf{m}^l{}_{j^*} - \mathbf{q}^l\|_2 - r^l{}_{j^*}. \tag{2}$$

We then have the following inequality property:

Property 3 *Given a query point* $\mathbf{q}$ *and a sample point* $\mathbf{p}^*$*, the LB-distance between the level-l ancestor of* $\mathbf{p}^*$ *(that is,* $\langle \mathbf{m}^l{}_{j^*}\rangle$*) and the level-l projection of* $\mathbf{q}$ *is smaller than or equal to the distance between* $\mathbf{p}^*$ *and* $\mathbf{q}$*. That is,*

$$d_{LB}(\langle \mathbf{m}^l{}_{j^*}\rangle, \mathbf{q}^l) \leq \|\mathbf{p}^* - \mathbf{q}\|_2, \; l = 0, \ldots, L.$$

We know now from the above property that $d_{LB}(\langle \mathbf{m}^l{}_{j^*}\rangle, \mathbf{q}^l)$ is a lower bound of the distance $\|\mathbf{p}^* - \mathbf{q}\|_2$. Notice that the LB-distance is not a valid distance metric. A negative $d_{LB}(\langle \mathbf{m}^l{}_{j^*}\rangle, \mathbf{q}^l)$ implies that the query point $\mathbf{q}$ is located within the bounding sphere of $C^l{}_{j^*}$ centered at $\mathbf{m}^l{}_{j^*}$. Also, $d_{LB}(\langle \mathbf{m}^0{}_{j^*}\rangle, \mathbf{q}^0)$, $d_{LB}(\langle \mathbf{m}^1{}_{j^*}\rangle, \mathbf{q}^1)$, …, $d_{LB}(\langle \mathbf{m}^L{}_{j^*}\rangle, \mathbf{q}^L)$ is not necessarily (and not required to be) an ascending list of lower bounds of the distance between $\mathbf{p}^*$ and $\mathbf{q}$.

The internal node $\langle \mathbf{m}^l{}_{j^*}\rangle$ can have a number of descendants. Therefore, besides being a lower bound on the distance to $\mathbf{q}$ for any particular $\mathbf{p}^*$, $d_{LB}(\langle \mathbf{m}^l{}_{j^*}\rangle$ is also a lower bound on the distances from all the sample points in cluster $C^l{}_{j^*}$ containing $\mathbf{p}^*$ to $\mathbf{q}$. Hence, we have the following property:

Property 4 *Let* $\mathbf{q}$ *be a query point and* $\hat{\mathbf{p}}$ *be a sample point. For any internal node* $\langle \mathbf{m}^l{}_j\rangle$ *of the LB-tree, if*

$$d_{LB}(\langle \mathbf{m}^l{}_j\rangle, \mathbf{q}^l) > \|\hat{\mathbf{p}} - \mathbf{q}\|_2,$$

then, for every descendant leaf node $\langle \mathbf{p}'\rangle$ *of* $\langle \mathbf{m}^l{}_j\rangle$*, we have*

$$\|\mathbf{p}' - \mathbf{q}\|_2 > \|\hat{\mathbf{p}} - \mathbf{q}\|_2.$$

From Property 4, if the LB-distance of the internal node $\langle \mathbf{m}^l{}_j\rangle$ is already larger than the distance between $\hat{\mathbf{p}}$ and $\mathbf{q}$, all the descendant leaf nodes $\langle \mathbf{p}'\rangle$ of $\mathbf{m}^l{}_j$ can be eliminated in the search, since there is already a better candidate, $\hat{\mathbf{p}}$, which is closer to $\mathbf{q}$.

3 Winner-Update Search Strategy and the Proposed Algorithm

In our algorithm an LB-tree of $L+1$ levels has to be constructed using data set P before running the query process. For each query point $\mathbf{q}$, the goal of nearest neighbor search is to find the sample point $\hat{\mathbf{p}}$ in P such that the Euclidean distance $\|\hat{\mathbf{p}} - \mathbf{q}\|_2$ is minimum. According to Property 4, if at some point the LB-distance of an internal node $\langle \mathbf{m}^l{}_j \rangle$ is larger than the minimum distance between $\hat{\mathbf{p}}$ and $\mathbf{q}$, then the nearest neighbor cannot be in the descendant samples of node $\langle \mathbf{m}^l{}_j \rangle$. Hence, the costly calculation of their distances to $\mathbf{q}$ can be all saved by only calculating the less-expensive LB-distance of node $\langle \mathbf{m}^l{}_j \rangle$.

The above saving requires knowing the value $\|\hat{\mathbf{p}} - \mathbf{q}\|_2$, but it is unknown beforehand which sample point $\hat{\mathbf{p}}$ is. In fact, $\hat{\mathbf{p}}$ is exactly the nearest neighbor which we are looking for. To achieve the same calculation saving effect, we adopt the winner-update search strategy, which computes the lower bounds from the root node toward the leaf nodes while traversing the LB-tree. The LB-distances of the internal nodes is calculated starting from the top level down. Since the computation cost of the LB-distance is smaller at the upper level, and an upper-level node generally has more descendants, we can save more distance calculation if the LB-distance of an upper-level node is already larger than the minimum distance.

We now describe the winner-update search strategy that greatly reduces the number of LB-distance calculations. First, the LB-distances between $\mathbf{q}^0$ and all the level-0 nodes in the LB-tree are calculated using Equation (2). A heap data structure is then constructed using these level-0 nodes, $\langle \mathbf{m}^0{}_1 \rangle$, $\langle \mathbf{m}^0{}_2 \rangle$, $\ldots$, $\langle \mathbf{m}^0{}_{s^0} \rangle$, with the root node of the heap, $\langle \hat{\mathbf{p}} \rangle$, being the node having the minimum LB-distance. Then, we delete the node $\langle \hat{\mathbf{p}} \rangle$ and insert its children into the heap, calculating their LB-distances and rearranging the heap to maintain the heap property. This produces a new root node with the minimum LB-distance, which becomes the new $\langle \hat{\mathbf{p}} \rangle$. The procedure of deleting $\langle \hat{\mathbf{p}} \rangle$ and inserting its children is repeated until the dimension of $\langle \hat{\mathbf{p}} \rangle$, $dim(\langle \hat{\mathbf{p}} \rangle)$, is equal to d. At this time, with the node $\langle \hat{\mathbf{p}} \rangle$ being a leaf node containing a sample point, we have the minimum distance $\|\hat{\mathbf{p}} - \mathbf{q}\|_2$ in the heap. The nearest neighbor $\langle \hat{\mathbf{p}} \rangle$ is thus determined, since the lower bounds of the distances from all the other sample points to the query point $\mathbf{q}$ are already larger than $\|\hat{\mathbf{p}} - \mathbf{q}\|_2$.

Figure 5 illustrates three intermediate stages of the heap that are constructed during the search process, based on the LB-tree example shown in Figure 3. Given a query point $\mathbf{q}$, the LB-distances $d_{LB}(\langle \mathbf{m}^0{}_1 \rangle, \mathbf{q}^0)$ and $d_{LB}(\langle \mathbf{m}^0{}_2 \rangle, \mathbf{q}^0)$ for nodes $\langle \mathbf{m}^0{}_1 \rangle$ and $\langle \mathbf{m}^0{}_2 \rangle$ at level 0 are first calculated respectively and used to construct a heap, as shown in Figure 5(a). Suppose $d_{LB}(\langle \mathbf{m}^0{}_1 \rangle, \mathbf{q}^0)$ is 6

and $d_{LB}(\langle \mathbf{m}^0{}_2 \rangle, \mathbf{q}^0)$ is 2. At this point, node $\langle \mathbf{m}^0{}_2 \rangle$ is on top of the heap and will be replaced by its two children: nodes $\langle \mathbf{m}^1{}_2 \rangle$ and $\langle \mathbf{m}^1{}_3 \rangle$. Next, suppose $d_{LB}(\langle \mathbf{m}^1{}_2 \rangle, \mathbf{q}^1)$ is 8 and $d_{LB}(\langle \mathbf{m}^1{}_3 \rangle, \mathbf{q}^1)$ is 3. Then, the heap is rearranged to maintain the heap property, and node $\langle \mathbf{m}^1{}_3 \rangle$ will pop up to the top of the heap, as shown in Figure 5(b). Again, the new top node (that is, $\langle \mathbf{m}^1{}_3 \rangle$) is replaced by its children and the heap is rearranged according to the LB-distances of the nodes. Figure 5(c) illustrates the heap at this stage, where the LB-distances of the newly inserted nodes $\langle \mathbf{m}^2{}_4 \rangle$ and $\langle \mathbf{m}^2{}_5 \rangle$ are 9 and 4, respectively.

The proposed algorithm is summarized below:

Proposed Algorithm for Nearest Neighbor Search

/* Preprocessing Stage */
(1) Given a data set $P = \{\mathbf{p}_i \in R^d | i = 1, \ldots, s\}$
(2) Construct the LB-tree of $L + 1$ levels for P

/* Nearest Neighbor Search Stage */
(3) Given a query point $\mathbf{q} \in R^d$
(4) Construct the $(L + 1)$-level structure of $\mathbf{q}$
(5) Insert the root node of the LB-tree into an empty heap
(6) Let $\langle \hat{\mathbf{p}} \rangle$ be the root node of the heap
(7) **while** $dim(\langle \hat{\mathbf{p}} \rangle) < d$ **do**
(8) Delete node $\langle \hat{\mathbf{p}} \rangle$ from the heap
(9) Calculate the LB-distances for all the children of $\langle \hat{\mathbf{p}} \rangle$
(10) Insert all the children of $\langle \hat{\mathbf{p}} \rangle$ into the heap
(11) Rearrange the heap to maintain the heap property that the root node is the node having the minimum LB-distance
(12) Update $\langle \hat{\mathbf{p}} \rangle$ as the root node of the heap
(13) **endwhile**
(14) Output $\hat{\mathbf{p}}$

For conciseness of the above pseudo code, the heap is initialized as the dummy root node of the LB-tree, instead of the level-0 nodes. This will not affect the result since the dummy root node is replaced immediately in the first iteration of the loop by its children, that is, all the level-0 nodes.

Due to the adoption of the winner-update search strategy, which is actually the best-first search strategy, the proposed algorithm can be regarded as a special case of the A* algorithm [33]. In our algorithm the path cost term is always zero and the estimated distance to the goal node is the LB-distance, which is a lower bound of the distance from a sample point to the query point.

4 Other Query Types

With slight modification, the proposed algorithm can also speed up the following three search tasks: (i) the progressive search for k-nearest neighbors, (ii) the search for k-nearest neighbors within a specified distance threshold, and (iii) the search for neighbors that are close enough to the query, compared with the nearest neighbor.

4.1 Progressive Search for k-Nearest Neighbors

When the nearest neighbor is obtained by using the proposed algorithm, there may be some other candidates in the heap. Distance calculation for these candidates is partially performed and we can continue the search process to determine the next nearest neighbor without starting all over again. In general, we can easily extend the proposed algorithm to find the k-nearest neighbors, $1 < k \leq s$, in the following way. Once the nearest neighbor $\hat{\mathbf{p}}$ is obtained by using the algorithm described in Section 3, we can delete it from the heap and continue the process until the second nearest neighbor is obtained. By repeating the above procedure, one can obtain the third nearest neighbor, the fourth nearest neighbor, and so on, until all the desired k-nearest neighbors are obtained. The following pseudo code can be merged into the original algorithm, in the designated line number order, to provide k-nearest neighbors:

(6.5) **for** $loop = 1, 2, \ldots, k$

(15) Delete node $\langle \hat{\mathbf{p}} \rangle$ from the heap
(16) Rearrange the heap to maintain the heap property that the root node is the node having the minimum LB-distance
(17) Update $\langle \hat{\mathbf{p}} \rangle$ as the root node of the heap
(18) **endfor**

Notice that these k-nearest neighbors are provided incrementally. This feature is particularly useful when additional tests on the obtained nearest neighbors are required, and hence, the number k cannot be known before the query process begins. Hjaltason and Samet ascribed the capability of incremental k-nearest neighbor search to the heap (priority queue) employed in the algorithm [30]. Arya et al. [27] adopted a similar progressive approach which enumerates leaf cells of their BBD-tree in increasing order of distance from the query point and examines data points in the cells. Traditional methods such as [19] use an array to record the first k candidates of k-nearest neighbors during the search process. Each newly-computed distance is compared against the elements in the array and is substituted for the largest element in the array

that is larger than the newly-computed distance. After the k-nearest neighbors are determined and we find that more nearest neighbors are needed, the search process has to be started all over again with a larger k. As a result, there are wasteful, duplicated distance calculations.

4.2 Search for k-Nearest Neighbors within a Distance Threshold

In many pattern recognition applications, a query object is considered to be "recognized with high confidence" only when it is sufficiently close to an object in the data set. Therefore, the distance between the query point and its nearest neighbor should be smaller than a pre-specified distance threshold ε_T. For further speedup, the proposed algorithm can be easily extended to meet this requirement by adding the following two lines to the pseudo code of Section 3:

(7.5) **if** the LB-distance of $\langle \hat{\mathbf{p}} \rangle$ is larger than ε_T **stop**

(13.5) **if** the LB-distance of $\langle \hat{\mathbf{p}} \rangle$ is larger than ε_T **stop**

When the k-nearest neighbors within the distance threshold ε_T are needed, the additional pseudo code for providing k-nearest neighbors, given in Section 4.1, can also be added.

4.3 Search for Neighbors Close Enough to the Query Compared with the Nearest Neighbor

In some applications, all the points that are sufficiently close to the query point, compared with the nearest neighbor, should be considered as good matches. To achieve this goal, all the points of distance smaller than $(1 + r)\|\hat{\mathbf{p}}-\mathbf{q}\|_2$ have to be identified, where $\hat{\mathbf{p}}$ is the nearest neighbor and r is a small number. Our algorithm can be easily extended to provide this functionality. After the nearest neighbor $\hat{\mathbf{p}}$ and the minimum distance $\|\hat{\mathbf{p}}-\mathbf{q}\|_2$ are obtained, the methods described in Sections 4.1 and 4.2 can be used to provide all the points having distance smaller than ε_T, where k is set to be s and the threshold ε_T is set to be $(1 + r)\|\hat{\mathbf{p}} - \mathbf{q}\|_2$.

5 Construction of LB-Tree

The LB-tree plays an important role in our algorithm. It should be noted that there exists more than one method for constructing the LB-tree described in Section 2.2. Although many methods could be chosen for constructing the

LB-tree, some lead to better performance than others. Hence, it is desirable to construct a "good" LB-tree in view of the need for efficiency in nearest neighbor search. Since construction of the LB-tree is performed in the preprocessing stage, its computational cost is not a major concern here, hence the efficiency of the resulting LB-tree is more important than the speed of its construction.

To construct an LB-tree, the simplest way is to directly use the multilevel structures of the sample points without clustering. In this case, there are s nodes at each level l, $l = 0, \ldots, L$, in the LB-tree. Each node $\langle \mathbf{m}^l{}_i \rangle$, $i = 1, \ldots, s$, at level l contains exactly one level-l projection of a sample point, say $\mathbf{p}_i{}^l$. Here, the mean point $\mathbf{m}^l{}_i$ equals $\mathbf{p}_i{}^l$ and the radius $r^l{}_i$ is set to zero. All the internal nodes in the LB-tree thus constructed have only one child node, with the exception of the root, which has s child nodes.

Another method of LB-tree construction is to use k-means clustering method [3] to hierarchically cluster the sample points, similar to what was done in [19]. At level 0, all the sample points are partitioned into k disjoint clusters according to the distances between their level-0 projections. For each cluster at level 0, the mean point and the radius of the bounding sphere can be calculated and recorded by using the level-0 projections of the sample points that belong to the same cluster. Then, these sample points that belong to the same cluster at level 0 can be further partitioned into k disjoint sub-clusters according to the distances between their level-1 projections. After partitioning all the clusters at level 0, all the obtained sub-clusters constitute the nodes at level 1 of the LB-tree. This process is repeated for the succeeding levels until level-L is reached. The result is an LB-tree (but maybe not the best one), in which every internal node has k branches.

From Property 3 as defined in Section 2.2, given a query point $\mathbf{q}$, the LB-distance for each internal node (that is, the LB-distance between an internal node and the level-l projection of $\mathbf{q}$) is the lower bound of the distances between $\mathbf{q}$ and all the sample points contained in the descendant leaf nodes of this internal node. In order to obtain a tighter lower bound and thus, in order to reduce the number of distance calculations, the LB-distance of each internal node should be as large as possible, which requires *the radius of the bounding sphere, $r^l{}_j$, to be as small as possible* in consequence, according to Equation (2). From this perspective, it is suggestible to adopt the simplest construction method mentioned before, which directly uses the multilevel structures of the sample points, since the radius of each internal node is zero. However, the number of internal nodes is proportional to the amount of memory storage and computation of the LB-distance required, so we would also like *the number of internal nodes to be as small as possible.* Although the k-means clustering method can construct an LB-tree with fewer internal nodes, the radius of the bounding sphere may be very large since there is a

chance that sample points far away from each other are grouped into one cluster. As a result, the trade-off between the number of internal nodes and the radii of the associated bounding spheres needs to be taken into consideration when constructing an LB-tree.

In this work, we use an agglomerative hierarchical clustering technique [3,34] to construct the LB-tree, in which both the number of internal nodes and the associated radii can be kept small. Details are given below.

5.1 Multi-Dimensional Case for Each Level

Suppose that the level-l projections of all the sample points have been partitioned into s^l clusters, ${C^l}_j$, $j = 1, \ldots, s^l$. (For instance, the example shown in Figure 2 contains three clusters, ${C^1}_1$, ${C^1}_2$, and ${C^1}_3$, at level 1.) Each cluster ${C^l}_j$ at level l is to be further partitioned into sub-clusters independently. Notice that cluster ${C^l}_j$ is a set of level-l projections. Denote the members of ${C^l}_j$ as ${\mathbf{p}_{i_{lj}(k)}}^l$, $k = 1, 2, \ldots, n_{lj}$, where n_{lj} is the number of elements in ${C^l}_j$. Consider the example shown in Figure 2. For $l = 1$ and $j = 3$, we have ${C^1}_3 = \{{\mathbf{p}_3}^1, {\mathbf{p}_4}^1, {\mathbf{p}_7}^1\}$ where $n_{13} = 3$, $i_{13}(1) = 3$, $i_{13}(2) = 4$, and $i_{13}(3) = 7$. Then, by using the multilevel structures of $\mathbf{p}_{i_{lj}(k)}$, $k = 1, 2, \ldots, n_{lj}$, we denote the set of level-$(l + 1)$ projections $\{{\mathbf{p}_{i_{lj}(k)}}^{l+1} | k = 1, 2, \ldots, n_{lj}\}$ by ${S^{l+1}}_j$. For the above example ($l = 1$ and $j = 3$), we have ${S^2}_3 = \{{\mathbf{p}_3}^2, {\mathbf{p}_4}^2, {\mathbf{p}_7}^2\}$. Our approach is to partition ${S^{l+1}}_j$ into clusters by using an agglomerative method. In the example of Figure 2, ${S^2}_3$ is then partitioned into ${C^2}_4$ and ${C^2}_5$.

The agglomerative method begins with treating each point as a distinct cluster, and successively merges clusters together until a stopping criterion is satisfied [3]. There are two issues to be determined when adopting the agglomerative method. The first concerns how to choose clusters for merging, and the other is the stopping criterion. Suppose X and Y are two disjoint subsets of ${S^{l+1}}_j$. We define the between-cluster distance, ${d_{max}}^{l+1}(X, Y)$, of X and Y as the maximum Euclidean distance between every pair $(\mathbf{x}^{l+1}, \mathbf{y}^{l+1})$ of level-$(l + 1)$ projections, where $\mathbf{x}^{l+1} \in X$ and $\mathbf{y}^{l+1} \in Y$. That is,

$$ {d_{max}}^{l+1}(X, Y) = \max_{\mathbf{x}^{l+1} \in X, \mathbf{y}^{l+1} \in Y} \|\mathbf{x}^{l+1} - \mathbf{y}^{l+1}\|_2. $$

The pair of clusters with minimum ${d_{max}}^{l+1}$ is chosen for consideration to be merged because they are the closest clusters in the sense of ${d_{max}}^{l+1}$. The radius of the cluster obtained by merging this pair is more likely to be small.

For the stopping criterion, we use the *radius constraint* which requires that the radii of all clusters at level l are smaller than a pre-specified radius threshold ${r_T}^l$. When further cluster merging cannot satisfy the radius constraint, the

agglomerative procedure is terminated. In this way we can obtain a clustering result by gradually reducing the number of clusters via merging while the radius of each cluster gradually increases, approaching the radius threshold. If we raise the radius threshold, the number of clusters (and the resulted number of nodes) at level $(l+1)$ will decrease. By specifying a "good" radius threshold, the number of internal nodes and their associated radii reach a compromise and a good clustering result can be obtained.

For the set ${S^{l+1}}_j = \{{\mathbf{p}_{i_{lj}(k)}}^{l+1} | k = 1, 2, \ldots, n_{lj}\}$, we initially treat each of its member as a separate cluster. Then we calculate and sort the between-cluster distances ${d_{max}}^{l+1}$ for every pair of clusters. The pair of clusters with the minimum ${d_{max}}^{l+1}$ is chosen for consideration of merging. If ${d_{max}}^{l+1}$ of this pair is larger than twice the radius threshold ${r_T}^{l+1}$, the radius of the bounding sphere of the merged cluster will definitely be larger than ${r_T}^{l+1}$, and so violates the radius constraint. In this case, clustering can be terminated because no further merging can satisfy the radius constraint. Otherwise, we tentatively merge this pair of clusters by computing the mean of the merged cluster and the radius of its bounding sphere. If the newly-computed radius is indeed smaller than the radius threshold ${r_T}^{l+1}$, this pair of clusters will actually be merged. The between-cluster distances ${d_{max}}^{l+1}$ between this merged cluster and all other clusters must be updated accordingly. If the newly-computed radius is not smaller than the radius threshold, we do not merge this pair of clusters and instead choose the pair with the second minimum ${d_{max}}^{l+1}$ for consideration. This procedure is repeated until all the pairs are examined or the ${d_{max}}^{l+1}$ of the examined pair is larger than twice the radius threshold ${r_T}^{l+1}$.

All sample points whose level-$(l+1)$ projections are grouped into the same cluster at level $l+1$ can be further partitioned at level $l+2$ by using the same method presented above. This recursive clustering is applied until the bottom level is reached, where each sample point is treated as a separate cluster of zero radius. In this way we can construct an LB-tree satisfying the radius constraint with a specified radius threshold while trying to make the number of internal nodes as small as possible.

5.2 One-Dimensional Case for Level 0

If at level 0 the number of sample points, s, is large, the number of between-cluster distances for every pair of initial clusters can be very large ($O(s^2)$). Hence, the agglomerative clustering process can be very time-consuming at level 0. If there is only one dimension at level 0, as in this work, we can reduce this problem with the following method. The level-0 projections of all the sample points are first sorted. Then, consider only pairs of *neighboring* level-

0 projections for merging because the minimum ${d_{max}}^0$ appears only between the neighboring level-0 projections. In this way the number of the cluster pairs to be considered can be reduced from $O(s^2)$ to $O(s)$. When a pair of level-0 projections with minimum ${d_{max}}^0$ is merged, these two level-0 projections are replaced with their mean in the sorted list. This process is repeated until the radius of the bounding sphere (or rather, the bounding segment) for the best merging is larger than the radius threshold ${r_T}^0$.

5.3 Selection of the Radius Threshold

The radius threshold ${r_T}^l$ for each level has a great influence on the construction of the LB-tree and the resulting search efficiency. Remember that a tighter LB-distance can save more distance calculations. Toward the goal of achieving tighter LB-distances, we have to lower the radius threshold ${r_T}^l$ at level l in order to obtain smaller radii ${r^l}_j$ for all internal nodes at this level. However, a smaller radius threshold will in general result in more clusters, which tends to increase the computational cost of the proposed algorithm (because more LB-distances have to be calculated). This is the trade-off between choosing a smaller ${r^l}_j$ and choosing a smaller s^l, as mentioned in Section 5.1.

It is difficult to determine a good radius threshold beforehand because the choice depends on the distribution of the sample points. Therefore, instead of specifying a radius threshold ${r_T}^0$, the experiments shown in this paper specify the number of clusters s^0 at level 0, where $s^0 < s$. (For levels other than level 0, we specify radius thresholds as described below instead of specifying the number of clusters.) Hence, the stopping criterion at level 0 has to be modified accordingly in the following. All level-0 projections are merged agglomeratively until the number of clusters equals s^0. When the number of clusters reaches s^0, the radius of the latest merged cluster is recorded as ${r_T}^*$, and then used to determine the radius thresholds of the other levels. For each level l other than level 0, that is, $l = 1, \ldots, L-1$, the radius threshold ${r_T}^l$ can be determined based on ${r_T}^*$. (Note that there is no need to perform agglomerative clustering at level L because each cluster contains only one point at this level.) In this work, we simply use ${r_T}^*$ as the radius threshold at each level l, or ${r_T}^l = {r_T}^*$, $l = 1, \ldots, L-1$.

6 Data Transformation

This section explains how to further improve the efficiency of nearest neighbor search by applying data transformation. Recall that the Euclidean distance calculated at level l in the LB-tree is the distance in the subspace of the first

2^l dimensions. If these dimensions are not discriminative enough, meaning the projections of the sample points on this subspace are too close to each other, the distances may be almost identical for different samples calculated in this subspace, which will not help much in the search for nearest neighbor. To alleviate this problem, we apply transformation to the data points, transforming them into another space so that the anterior dimensions are likely to be more discriminative than the posterior dimensions. The transformation will affect efficiency but not the final search result, for the Euclidean distances calculated in both space should be the same. Moreover, because this transformation is also applied to the query points during the query process, it should be computationally inexpensive. The pseudo code for data transformation is as follows, which is to be joined with the algorithm in Section 3:

(1.5) Transform each sample point $\mathbf{p}_i$, $i = 1, \ldots, s$

(3.5) Transform the query point $\mathbf{q}$

Depending on the characteristics of the data, one of the following two types of data transformation can be used. Wavelet transform with orthonormal basis [35] is applied when the data point represents an autocorrelated signal, like an audio signal or an image block. The basis has to be orthonormal to preserve Euclidean distances. Here we adopt Haar wavelets for transforming the autocorrelated data, which is then represented in one of its multiple resolutions in each level in the multilevel structure. Readers are referred to [35] for computation method of Haar wavelets as well as the proof that Haar wavelets form an orthonormal basis.

Another type of data transformation is the principal component analysis (PCA). PCA finds a set of vectors ordered in their ability to account for the variation of data projected onto those vectors. The data point is transformed onto the space spanned by this set of vectors so that the anterior dimensions become more discriminative. This transformation is particularly useful for object recognition where not all features are equally important.

7 Experimental Results

In this section we show some experimental results of four algorithms: the exhaustive search algorithm (ES), the searching-by-slicing algorithm (SBS) proposed by Nene and Nayar [17], the BBD tree algorithm (BBDT) proposed by Arya et al. [27], and the lower bound tree algorithm (LBT) proposed in this paper. We obtained via FTP the software of the SBS algorithm and the BBDT algorithm implemented by Nene and Nayar [17] and Arya et al. [27], respectively. In SBS, the initial distance threshold is set to be 0.1. To guarantee

that the nearest neighbor can always be found, this threshold is enlarged gradually by adding 0.1 each time no point is found, as recommended in [17]. Remember that the BBDT algorithm can find the $(1+r)$-approximate nearest neighbor within a factor of $(1+r)$ of the distance between the query point and its exact nearest neighbor. To guarantee that the exact nearest neighbor can be found, we set the parameter r to be 0 in the software. For the the ES algorithm and the LBT algorithm, we have implemented in C programming language.

There are three different kinds of data distribution that we used to examine the efficiency for these algorithms, including a computer-generated set of autocorrelated data (Section 7.1), a computer-generated set of clustered Gaussian data (Section 7.2), and a real data set acquired from an object recognition system (Sections 7.3, 7.4, and 7.5). The experiments were performed on a PC with a Pentium III 700 MHz CPU. To compare the efficiency of different algorithms, we use the execution time instead of the number of distances calculated for the following two reasons. First, the insertion and deletion of an element in the heap, rearranging the heap, and updating node $\langle \hat{\mathbf{p}} \rangle$ results in our algorithm having some overhead. Second, the computational cost of the LB-distance of a node differs at different levels. To be specific, the computational cost of the LB-distance for nodes increases from the top level to the bottom level of the LB-tree.

7.1 Experiments on Autocorrelated Data

We now demonstrate the efficiency of the proposed algorithms by showing the result of three experiments as the following three factors vary: the number of sample points in the data set, s; the dimensionality of the underlying space, d; and the average of the minimum distances between query points and their nearest neighbors, $\overline{\varepsilon_{min}}$. Autocorrelated data points were randomly generated to simulate real signals. For each data point, the value of its first dimension was chosen from a uniform distribution with extent $[-1, 1]$, and the value of each subsequent dimension was assigned the value of the previous dimension plus normally distributed noise with zero mean and standard deviation 0.1. Beyond the extent $[-1, 1]$, the value of each dimension was truncated. In order to see how data transformation affects the search efficiency for autocorrelated data, we performed the nearest neighbor search twice for the SBS, BBDT, and LBT algorithms, with Haar transform applied to each data point only in the second time. The LB-tree was constructed with its number of clusters at level 0 specified as 45.

In the first experiment, we probed the algorithm efficiency by varying the number of sample points, s, in the data set. Seven data sets of s sample points,

s = 800, 1600, 3200, ..., 51200, were generated using the random process described above. The dimension of the underlying space, d, was 32. Constructing the LB-tree spent 0.1, 0.5, 1.6, 7.8, 67.5, 730.8, and 6434 seconds for each data set, respectively. Another set containing 100,000 query points were also generated using the same random process, and nearest neighbor search was then performed for each query point. Figure 6 shows the mean query time for each algorithm, where both the Haar transform, if applied, and the search process were taken into account. It is apparent in Figure 6 that the search efficiency of the SBS, BBDT, and LBT algorithms (without Haar transform), i.e., "SBS", "BBDT", and "LBT", can be significantly improved by applying the Haar transform, as denoted by "SBS+Haar", "BBDT+Haar", and "LBT+Haar". This clearly demonstrates that the Haar transform can help to reduce more computational cost when the data set consists of autocorrelated data. Among all the algorithms in this experiment, the proposed LBT algorithm (the "LBT+Haar" case) is the fastest one, which is 12.2 and 56.2 times faster than the ES algorithm, when s is 800 and 51,200, respectively. When s increases from 800 to 51,200, there are more sample points scattered in the fixed space, so the average minimum distance, $\overline{\varepsilon_{min}}$, decreases from 0.73 to 0.53. When the minimum distance is smaller, the LB-distance is then more likely to be larger than the minimum distance of the query point $\mathbf{q}$ to its nearest neighbor $\hat{\mathbf{p}}$, according to Property 4. That is, more distance calculations can be avoided if $\overline{\varepsilon_{min}}$ is smaller, which is why the speedup factor increases as s increases.

In the second experiment, we vary the dimensionality, d, of the underlying space. Eight data sets of 10,000 sample points, where dimension, d = 2, 4, 8, ..., 256, respectively, were generated. The construction time of the LB-tree was 3.1, 24.9, 26.7, 27.3, 28.5, 35.5, 37.3, and 51.8 seconds for each data set, respectively. The same random process was also used to generate eight corresponding sets of 100,000 query points, with matched dimensions $d = 2, 4, 8, \ldots, 256$. Figure 7 shows that the Haar transform can improve the search efficiency, particularly when d is large. The proposed LBT algorithm (the "LBT+Haar" case) outperforms the other algorithms when d is larger than 4. Interestingly note that our algorithm does not suffer from the curse of dimensionality for autocorrelated data like the k-dimensional binary search tree algorithm does, as reported in [17,16]. In fact, the computational speedup of the proposed algorithm (the "LBT+Haar" case) over the ES algorithm rises from 5.6 to 64.1 as d increases from 2 to 256. The increase of d also increases the level number of the multilevel structure and of the constructed LB-tree. Using the Haar transform causes the anterior dimensions to contain more significant components of the autocorrelated data, and so the lower bound of the distance can be tighter when calculated at the upper level. Distance calculation can therefore be avoided for more sample points by calculating only the LB-distances of a few of their upper-level ancestors, with exception of a few tough competitors. Without applying Haar transform (i.e., the "LBT" case),

each dimension of the data point is equally significant, and so the LB-distance at the lower level needs to be calculated to determine the nearest neighbor, which requires more computation and degrades performance. In addition, data transformation causes the agglomerative clustering from top to bottom to be more effective because the anterior dimensions contain more significant components. There are more internal nodes for the "LBT" case compared to that of the "LBT+Haar" case, and thus efficiency is reduced. The increase of d amplifies this phenomenon, which results in the dramatic drop of the speedup factor for the non-transform case, but not for the transform case.

The third experiment demonstrates the efficiency of the algorithms with respect to $\overline{\varepsilon_{min}}$. We generated a data set of $10,000$ sample points in a space of dimension $d = 32$, where each sample point was then used to generate a query point by adding a uniformly distributed noise with extent $[-e, e]$ to each coordinate. As a result, when e is large the distance between the query point and its nearest neighbor tends to be large as well. In this case, the construction time of the LB-tree is 29.6 seconds. In this experiment eight sets of $10,000$ query points are generated, with $e = 0.01, 0.02, 0.04, \ldots, 1.28$. The mean query time versus the mean of the minimum distances, $\overline{\varepsilon_{min}}$, is compared among different algorithms in Figure 8. Again, the Haar transform improves the search efficiency and the proposed LBT algorithm (the "LBT+Haar" case) outperforms the other algorithms. As e increases from 0.01 to 1.28, $\overline{\varepsilon_{min}}$ increases from 0.033 to 3.838. The increase in the computational cost of the LBT algorithm is due to the fact that when the minimum distance of the nearest neighbor is already very large, the LB-distance is less likely to be larger than the minimum distance, so less distance calculation can be saved. The speedup factor of the LBT algorithm (the "LBT+Haar" case), compared with the ES algorithm, decreases from 570.4 to 0.63 in this case. Notice that when the speedup factor becomes 0.63, the noise extent, $[-1.28, 1.28]$, is larger than the data extent, $[-1, 1]$. $\overline{\varepsilon_{min}}$ is usually relatively small for most applications, and therefore the case when the LBT algorithm does not outperform the ES algorithm, shown on the right part in Figure 8, does not likely happen.

7.2 Experiments on Clustered Gaussian Data

This section shows the experimental results when the sample point set consists of clustered Gaussian data, which was generated to simulate an object database. We first randomly chose 100 cluster center points in a 32-dimensional space. For each cluster center point, the value of each dimension was randomly generated from a uniform distribution with extent $[-1, 1]$. Then, we generated 100 sample points for each cluster. Each sample point was randomly chosen from a Gaussian distribution with standard deviation σ around the cluster center point. That is, the value of each dimension of a sample point was as-

signed the value of the corresponding dimension of the cluster center point added by normally distributed noise with zero mean and standard deviation σ. We obtained a set of $10,000$ sample points in this way. The LB-tree construction time was 78.7, 80.5, 98.7, 114, and 89.7 seconds, respectively, as σ ranging from 0.02, 0.04, ..., up to 0.1.

Around each of the same 100 cluster center points, we randomly chose another $1,000$ data points from the same Gaussian distribution with standard deviation σ. These $100,000$ points constituted the set of query points in the nearest neighbor search process. We totally generated five sets of sample points and query points with different standard deviation σ ranging from 0.02 up to 0.1. Table 1 shows the mean query time of nearest neighbor search by using the ES, SBS, BBDT, and LBT algorithms. Numbers in parentheses denote the speedup factor compared with the ES algorithm. Search efficiency of the proposed LBT algorithm is the best, particularly when the clusters are compact (i.e., σ is small). The reason is that the minimum distance from the query point to its nearest neighbor tends to be smaller, compares with the distances from the query to the points in different clusters, when the clusters are more compact.

7.3 Experiments on an Object Recognition Database

The database adopted in the experiments described here is the same as those in [1,17], which was generated from 72 images of an object taken at different poses for a total of 100 objects. Each of these $7,200$ 128×128 images was represented in vector form, and each vector was normalized to unit length. An eigenspace of dimension 35 can be computed from those normalized vectors, so that by projecting onto the eigenspace, each vector can then be compressed from $16,384$ dimensions to 35 dimensions. In the eigenspace, the manifold for each object can be constructed using the 72 vectors belonging to the object. Each of the 100 manifolds was sampled to obtain 360 vectors, resulting in a total of $s = 36,000$ sampled vectors constituting the data set, where each sample point has dimension $d = 35$.

To generate the set of query points, we first uniformly sample the manifolds by sampling each of the 100 manifolds at $3,600$ equally spaced positions. Then we add to each coordinate a uniformly distributed noise with extent $[-e, e]$. This yields a set of $360,000$ query points.

The ES, SBS, BBDT, and LBT algorithms were used to perform the nearest neighbor search. The initial distance threshold of the SBS algorithm was selected to be 0.035 in this experiment. Table 2 shows the mean query time for these algorithms when the noise extent e is 0.005, 0.01, and 0.015. Numbers

in parentheses denote the speedup factor compared with the ES algorithm. In this case the proposed LBT algorithm can tremendously speed up the nearest neighbor search process. When e is 0.005, the LBT algorithm is $1,088$ times faster than the ES algorithm. This performance is roughly 13 times faster than the result attained by the SBS algorithm and is roughly 1.7 times faster than the BBDT algorithm. Furthermore, the speedup factors of the LBT algorithm compared with the SBS and BBDT algorithms rise as the noise extent e rises. The construction time of the LB-tree is $11,679$ seconds using those $36,000$ sample points of dimension 35, and the number of clusters, s^l, at level l of the LB-tree are $s^l = 20, 245, 2456, 4684, 5716, 7019, 36000$, $l = 0, 1, \ldots, 6$. In this case, although the construction time is acceptable, more work should be done when dealing with a large sample point set to improve the efficiency of the LB-tree construction. The average of the minimum distances of all the sample points to their nearest neighbors is 0.017376.

7.4 Experiments for k-Nearest Neighbor Search

This section presents the experiments for k-nearest neighbor search using the BBDT algorithm and the LBT algorithm modified as described in Section 4.1. These experiments were performed with the object recognition database described in Section 7.3, and the same LB-tree and query point set with the noise extent $e = 0.005$ were used. However, instead of searching for only the single nearest neighbor, we searched for the k-nearest neighbors of each query point. Table 3 illustrates the mean query time for the k-nearest neighbor search, $k = 2, 4, \ldots, 20$. As k rises to 20, the mean query time using the LBT algorithm increases to 0.408 ms, which is about 8.9 times larger than that for 1-nearest neighbor search. When the BBDT algorithm is applied, the mean query time increases to 2.251 ms as k rises to 20. That is, the mean query time of 20-nearest neighbor search is about 28.5 times larger than that for 1-nearest neighbor search by using the BBDT algorithm. This concludes that there exists extra advantage if the proposed LBT algorithm is adopted for k-nearest neighbor search.

7.5 Experiments of Searching for k-Nearest Neighbors within a Distance Threshold

This section presents the experiments for the LBT algorithm that is modified as described in Section 4.2. Again, the same LB-tree and query point set described in Section 7.3 were used. For each query point, at most the first 20 of its nearest neighbors within the distance threshold ε_T were obtained. As ε_T rises from 0 to 0.108 ($\varepsilon_T = 0$ implies the requirement for a perfect match), the

mean query time goes from 0.023 ms to 0.193 ms, as shown in Table 4. As can be expected, larger ε_T will result in more neighbors obtained, and hence, more computation time. In this experiment, average number of obtained neighbors for all query points increases from 0 to 17.4.

8 Conclusions

In this paper we have proposed a fast algorithm for nearest neighbor search. By creating an LB-tree using the agglomerative clustering technique and then traversing the tree using the winner-update search strategy, we can efficiently find the exact nearest neighbor. To further speedup the search process, some data transformation is applied to sample points and query points, such as Haar transform (for autocorrelated data) and PCA (for general object recognition data). Moreover, the proposed algorithm can be easily extended to provide k-nearest neighbors progressively, nearest neighbors within a specified distance threshold, and close-enough neighbors compared with the nearest neighbor, respectively.

From our experiments, the search process is dramatically accelerated using the proposed algorithm, especially when the distance of the query point to its nearest neighbor is relatively small compared with its distance to most other sample points. Our algorithm is particularly advantageous in many object recognition applications, where a query point of an object is close to the sample points of the same object, but is far from the sample points of other objects. In this paper we applied our algorithm to the object recognition database used in [1,17], and the result is about five hundred to one thousand times faster than the exhaustive search. In addition, we believe that the proposed algorithm can be very helpful in applications where each sample point represents an autocorrelated signal, like applications concerning content-based retrieval from a large audio, image, or video database, as those in [9,36]. The dimension d and the number of sample points s in these applications are both large, and hence, our algorithm will become extremely appealing.

Acknowledgements

The authors would like to thank the helpful comments and suggestions given by the reviewers. This work was supported in part by the Ministry of Economic Affairs, Taiwan, under Grants 93-EC-17-A-02-S1-032 and 94-EC-17-A-02-S1-032.

References

[1] Hiroshi Murase and Shree K. Nayar, "Visual learning and recognition of 3-D objects from appearance," *International Journal of Computer Vision*, vol. 14, pp. 5–24, 1995.

[2] Trevor Hastie and Robert Tibshirani, "Discriminant adaptive nearest neighbor classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 6, pp. 607–616, 1996.

[3] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn, "Data clustering: A review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 264–323, 1999.

[4] Carlo Tomasi and Roberto Manduchi, "Stereo matching as a nearest-neighbor problem," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 3, pp. 333–340, 1998.

[5] Yong-Sheng Chen, Yi-Ping Hung, and Chiou-Shann Fuh, "Fast block matching algorithm based on the winner-update strategy," *IEEE Transactions on Image Processing*, vol. 10, no. 8, pp. 1212–1222, Aug. 2001.

[6] Chang-Hsing Lee and Ling-Hwei Chen, "A fast search algorithm for vector quantization using mean pyramids of codewords," *IEEE Transactions on Communications*, vol. 43, no. 2/3/4, pp. 1697–1702, 1995.

[7] Chaur-Heh Hsieh and Yong-Jzu Liu, "Fast search algorithms for vector quantization of images using multiple triangle inequalities and wavelet transform," *IEEE Transactions on Image Processing*, vol. 9, no. 3, pp. 321–328, 2000.

[8] Li-Yi Wei and Marc Levoy, "Fast texture synthesis using tree-structured vector quantization," in *Proceedings of SIGGRAPH*, New Orleans, Louisiana, July 2000, pp. 479–488.

[9] Myron Flickner, Harpreet Sawhney, Wayne Niblack, Jonathan Ashley, Qian Huang, Byron Dom, Monika Gorkani, Jim Hafner, Denis Lee, Dragutin Petkovic, David Steele, and Peter Yanker, "Query by image and video content: The QBIC sytstem," *IEEE Computer*, vol. 28, no. 9, pp. 23–32, 1995.

[10] Stefan Berchtold, Christian Böhm, Bernhard Braunmüller, Daniel A. Keim, and Hans-Peter Kriegel, "Fast parallel similarity search in multimedia databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, USA, May 1997, pp. 1–12.

[11] Jon Louis Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[12] Antonin Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Boston, MA, June 1984, pp. 47–57.

[13] Norio Katayama and Shin'ichi Satoh, "The SR-tree: An index structure for high-dimensional nearest neighbor queries," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, USA, May 1997, pp. 369–380.

[14] David A. White and Ramesh Jain, "Similarity indexing with the SS-tree," in *Proceedings of the International Conference on Data Engineering*, New Orleans, Louisiana, Feb. 1996, pp. 516–523.

[15] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel, "The pyramid-technique: Towards breaking the curse of dimensionality," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, June 1998, pp. 142–153.

[16] Stefan Berchtold, Daniel A. Keim, Hans-Peter Kriegel, and Thomas Seidl, "Indexing the solution space: A new technique for nearest neighbor search in high-dimensional space," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 1, pp. 45–57, 2000.

[17] Sameer A. Nene and Shree K. Nayar, "A simple algorithm for nearest neighbor search in high dimensions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 9, pp. 989–1003, Sept. 1997.

[18] V. Ramasubramanian and Kuldip K. Paliwal, "Fast nearest-neighbor search algorithms based on approximation-elimination search," *Pattern Recognition*, vol. 33, no. 9, pp. 1497–1510, 2000.

[19] Keinosuke Fukunaga and Patrenahalli M. Narendra, "A branch and bound algorithm for computing k-nearest neighbors," *IEEE Transactions on Computers*, vol. 24, pp. 750–753, 1975.

[20] Sergey Brin, "Near neighbor search in large metric spaces," in *Proceedings of the International Conference on Very Large Data Bases*, Zurich, Switzerland, Sept. 1995, pp. 574–584.

[21] Enrique Vidal, "New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (AESA)," *Pattern Recognition Letters*, vol. 15, no. 1, pp. 1–7, Jan. 1994.

[22] Jerome H. Friedman, Forest Baskett, and Leonard J. Shustek, "An algorithm for finding nearest neighbors," *IEEE Transactions on Computers*, vol. 24, pp. 1000–1006, 1975.

[23] Mohammad Reza Soleymani and Salvatore D. Morgera, "An efficient nearest neighbor search method," *IEEE Transactions on Communications*, vol. COM-35, no. 6, pp. 677–679, 1987.

[24] Abdelhamid Djouadi and Essaid Bouktache, "A fast algorithm for the nearest-neighbor classifier," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 3, pp. 277–282, 1997.

[25] Eel-Wan Lee and Soo-Ik Chae, "Fast design of reduced-complexity nearest-neighbor classifiers using triangular inequality," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 567–571, 1998.

[26] James McNames, "A fast nearest-neighbor algorithm based on a principal axis search tree," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 9, pp. 964–976, Sept. 2001.

[27] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," *Journal of the ACM*, vol. 45, no. 6, pp. 891–923, 1998.

[28] András Faragó, Tamás Linder, and Gábor Lugosi, "Fast nearest-neighbor search in dissimilarity spaces," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15, no. 9, pp. 957–962, 1993.

[29] Ronald Fagin and Larry Stockmeyer, "Relaxing the triangle inequality in pattern matching," *International Journal of Computer Vision*, vol. 28, no. 3, pp. 219–231, 1998.

[30] Gísli R. Hjaltason and Hanan Samet, "Distance browsing in spatial databases," *ACM Transactions on Database Systems*, vol. 24, no. 2, pp. 265–318, 1999.

[31] Yong-Sheng Chen, Yi-Ping Hung, and Chiou-Shann Fuh, "Winner-update algorithm for nearest neighbor search," in *Proceedings of the International Conference on Pattern Recognition*, Barcelona, Spain, Sept. 2000, vol. 2, pp. 708–711.

[32] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos, "The TV-tree: An index structure for high-dimensional data," *The VLDB Journal*, vol. 3, no. 4, pp. 517–542, 1994.

[33] Patrick Henry Winston, *Artificial Intelligence*, Addison-Wesley, Reading, Massachusetts, third edition, 1992.

[34] Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*, John Wiley & Sons, New York, second edition, 2001.

[35] Gilbert Strang and Truong Nguyen, *Wavelets and Filter Banks*, Wellesley-Cambridge Press, Wellesley, Massachusetts, 1996.

[36] Howard D. Wactlar, Takeo Kanade, Michael A. Smith, and Scott M. Stevens, "Intelligent access to digital video: Informedia project," *IEEE Computer*, vol. 29, no. 5, pp. 46–52, 1996.

Biographical Sketch

About the Author—YONG-SHENG CHEN received his B.S. degree in computer and information science from National Chiao Tung University, Taiwan, in 1993. He received an M.S. degree and a Ph.D. degree in computer science and information engineering from National Taiwan University, Taiwan, in 1995 and 2001, respectively. He is currently an assistant professor in the Department of Computer Science, National Chiao Tung University, Taiwan. His research interests include biomedical signal processing, medical image processing, and computer vision.

About the Author—YI-PING HUNG received his B.Sc. in Electrical Engineering from the National Taiwan University in 1982. He received an M.Sc. from the Division of Engineering, an M.Sc. from the Division of Applied Mathematics, and a Ph.D. from the Division of Engineering, all at Brown University, in 1987, 1988 and 1990, respectively. He is currently a professor in the Graduate Institute of Networking and Multimedia, and in the Department of Computer Science and Information Engineering, both at the National Taiwan University. From 1990 to 2002, he was with the Institute of Information Science, Academia Sinica, Taiwan, where he became a tenured research fellow in 1997 and is now an adjunct research fellow. He received the Young Researcher Publication Award from Academia Sinica in 1997. His current research interests include computer vision, pattern recognition, image processing, virtual reality, multimedia and human-computer interaction.

About the Author—TING-FANG YEN received her B.S. in Computer Science and Information Engineering from National Chiao Tung University, Taiwan, in 2004. She is currently a graduate student in Electrical and Computer Engineering at Carnegie Mellon University. Ting-Fang's current research mainly focus on software security, including issues in exploit detection and recovery, and evaluating the effect of software diversity in preventing widespread attacks.

About the Author—CHIOU-SHANN FUH received the BS degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1983, the MS degree in computer science from the Pennsylvania State University, University Park, PA, in 1987, and the PhD degree in computer science from Harvard University, Cambridge, MA, in 1992. He was with AT&T Bell Laboratories and engaged in performance monitoring of switching networks from 1992 to 1993. He was an associate professor in Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan from 1993 to 2000 and then promoted to a full professor. His current research interests include digital image processing, computer vision, pattern recognition, mathematical morphology, and their applications to defect inspection, industrial automation, digital still camera, digital

video camcorder, and camera module such as color interpolation, auto exposure, auto focus, auto white balance, color calibration, and color management.

Figure Captions

Fig. 1. An example of the 4-level structure of the point $\mathbf{p}$, where $\mathbf{p} \in R^8$.

Fig. 2. An example of hierarchical construction of the LB-tree. All the points in the same dark region are determined agglomeratively and are grouped into a cluster. Notice that each point is transposed in order to fit into the limited space.

Fig. 3. An example of the LB-tree.

Fig. 4. Illustration of the distance inequality of Equation (1).

Fig. 5. Three intermediate stages of the heap. (a) Given a query point $\mathbf{q}$, the LB-distances for nodes $\langle \mathbf{m}^0{}_1 \rangle$ and $\langle \mathbf{m}^0{}_2 \rangle$ at level 0 are calculated and used to construct a heap. (b) The node $\langle \mathbf{m}^0{}_2 \rangle$ in (a) has the smaller LB-distance and is replaced by its children: nodes $\langle \mathbf{m}^1{}_2 \rangle$ and $\langle \mathbf{m}^1{}_3 \rangle$. (c) The node $\langle \mathbf{m}^1{}_3 \rangle$ in (b) has the smallest LB-distance and is replaced by nodes $\langle \mathbf{m}^2{}_4 \rangle$ and $\langle \mathbf{m}^2{}_5 \rangle$.

Fig. 6. Mean query time versus size, s, of the sample point set ($d = 32$).

Fig. 7. Mean query time versus dimension of the underlying space, d ($s = 10,000$).

Fig. 8. Mean query time versus mean of the minimum distances, $\overline{\varepsilon_{min}}$ ($s = 10,000$, $d = 32$).

Table 1
Efficiency comparison for clustered Gaussian data with different σ

Algorithm	σ=0.02	σ=0.04	σ=0.06	σ=0.08	σ=0.1
ES	11.938 ms.				
SBS	0.619 ms. (19)	0.679 ms. (18)	0.710 ms. (17)	1.024 ms. (12)	1.916 ms. (6)
BBDT	0.236 ms. (51)	0.243 ms. (49)	0.249 ms. (48)	0.265 ms. (45)	0.289 ms. (41)
LBT	0.047 ms. (254)	0.052 ms. (230)	0.077 ms. (155)	0.082 ms. (146)	0.115 ms. (104)

Table 2
Efficiency comparison for an object recognition database

Algorithm	e=0.005	e=0.01	e=0.015
ES	50.048 ms.		
SBS	0.613 ms. (82)	1.096 ms. (46)	2.161 ms. (23)
BBDT	0.079 ms. (634)	0.164 ms. (305)	0.281 ms. (178)
LBT	0.046 ms. (1088)	0.072 ms. (695)	0.095 ms. (527)

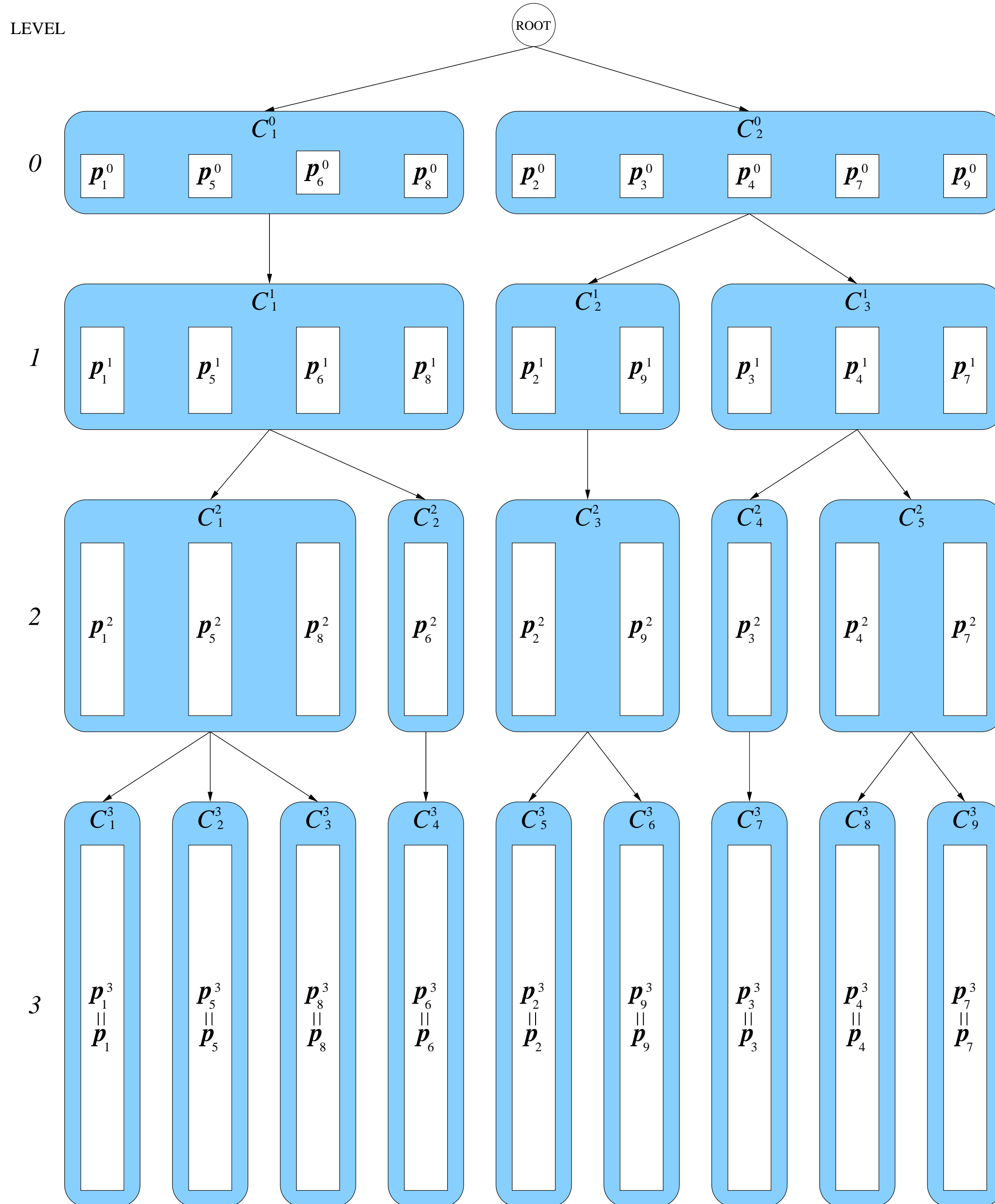
Table 3
Mean query time (in ms.) for k-nearest neighbor search

	k									
Algorithm	2	4	6	8	10	12	14	16	18	20
BBDT	0.087	0.121	0.204	0.363	0.587	0.875	1.192	1.532	1.885	2.251
LBT	0.053	0.069	0.091	0.121	0.158	0.202	0.252	0.302	0.356	0.408

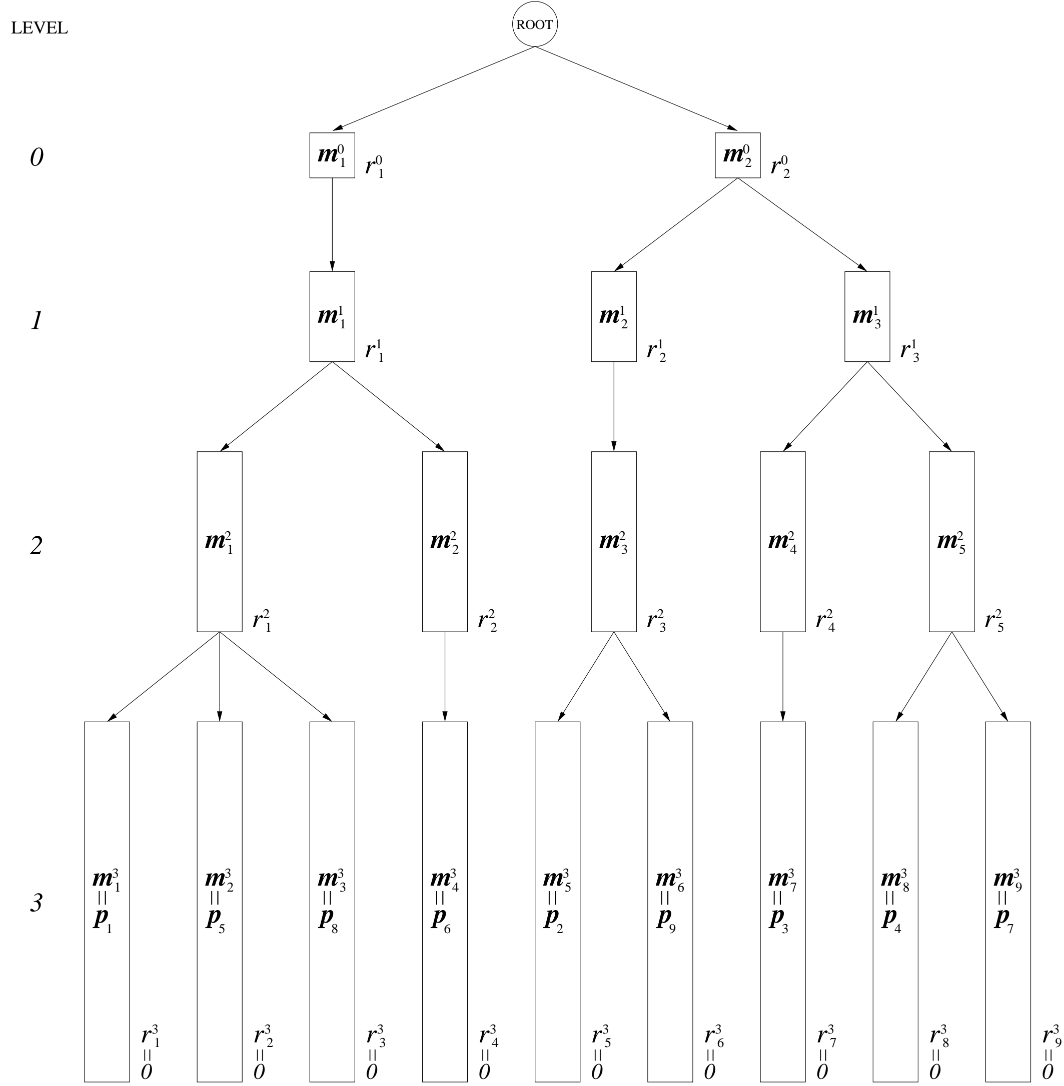
Table 4
Mean query time (in ms.) for ε_T-nearest neighbor search.

	ε_T									
Algorithm	0	0.012	0.024	0.036	0.048	0.060	0.072	0.084	0.096	0.108
LBT	0.023	0.040	0.060	0.078	0.096	0.114	0.134	0.154	0.174	0.193

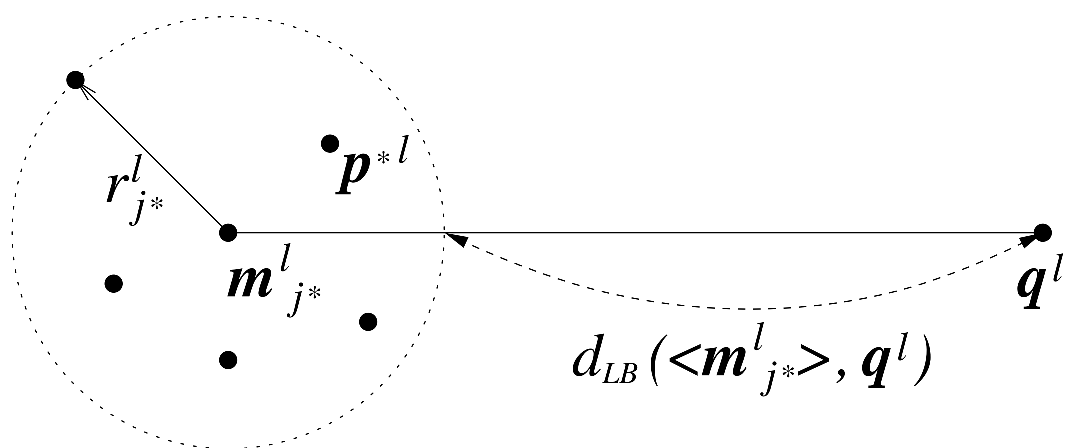
$\boldsymbol{p}^0$	p_1							
$\boldsymbol{p}^1$	p_1	p_2						
$\boldsymbol{p}^2$	p_1	p_2	p_3	p_4				
$\boldsymbol{p}^3$	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8

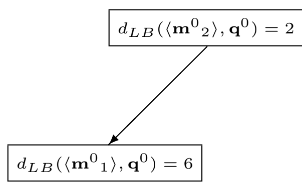


LEVEL
ROOT
0
C^0_1
p^0_1 p^0_5 p^0_6 p^0_8
C^0_2
p^0_2 p^0_3 p^0_4 p^0_7 p^0_9
1
C^1_1
p^1_1 p^1_5 p^1_6 p^1_8
C^1_2
p^1_2 p^1_9
C^1_3
p^1_3 p^1_4 p^1_7
2
C^2_1
p^2_1 p^2_5 p^2_8
C^2_2
p^2_6
C^2_3
p^2_2 p^2_9
C^2_4
p^2_3
C^2_5
p^2_4 p^2_7
3
C^3_1 C^3_2 C^3_3 C^3_4 C^3_5 C^3_6 C^3_7 C^3_8 C^3_9
$p^3_1 = p_1$
$p^3_5 = p_5$
$p^3_8 = p_8$
$p^3_6 = p_6$
$p^3_2 = p_2$
$p^3_9 = p_9$
$p^3_3 = p_3$
$p^3_4 = p_4$
$p^3_7 = p_7$

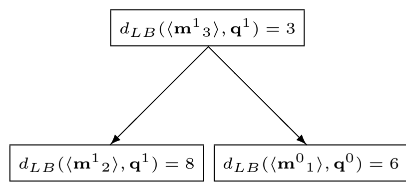


LEVEL
ROOT
0
1
2
3
$\boldsymbol{m}_1^0$ r_1^0
$\boldsymbol{m}_2^0$ r_2^0
$\boldsymbol{m}_1^1$ r_1^1
$\boldsymbol{m}_2^1$ r_2^1
$\boldsymbol{m}_3^1$ r_3^1
$\boldsymbol{m}_1^2$ r_1^2
$\boldsymbol{m}_2^2$ r_2^2
$\boldsymbol{m}_3^2$ r_3^2
$\boldsymbol{m}_4^2$ r_4^2
$\boldsymbol{m}_5^2$ r_5^2
$\boldsymbol{m}_1^3 = \boldsymbol{p}_1$ $r_1^3 = 0$
$\boldsymbol{m}_2^3 = \boldsymbol{p}_5$ $r_2^3 = 0$
$\boldsymbol{m}_3^3 = \boldsymbol{p}_8$ $r_3^3 = 0$
$\boldsymbol{m}_4^3 = \boldsymbol{p}_6$ $r_4^3 = 0$
$\boldsymbol{m}_5^3 = \boldsymbol{p}_2$ $r_5^3 = 0$
$\boldsymbol{m}_6^3 = \boldsymbol{p}_9$ $r_6^3 = 0$
$\boldsymbol{m}_7^3 = \boldsymbol{p}_3$ $r_7^3 = 0$
$\boldsymbol{m}_8^3 = \boldsymbol{p}_4$ $r_8^3 = 0$
$\boldsymbol{m}_9^3 = \boldsymbol{p}_7$ $r_9^3 = 0$

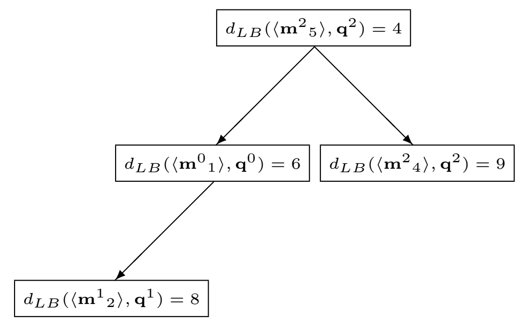

r^l_{j*}
$\boldsymbol{p}^{*l}$
$\boldsymbol{m}^l_{j*}$
$\boldsymbol{q}^l$
$d_{LB}(<\boldsymbol{m}^l_{j*}>, \boldsymbol{q}^l)$



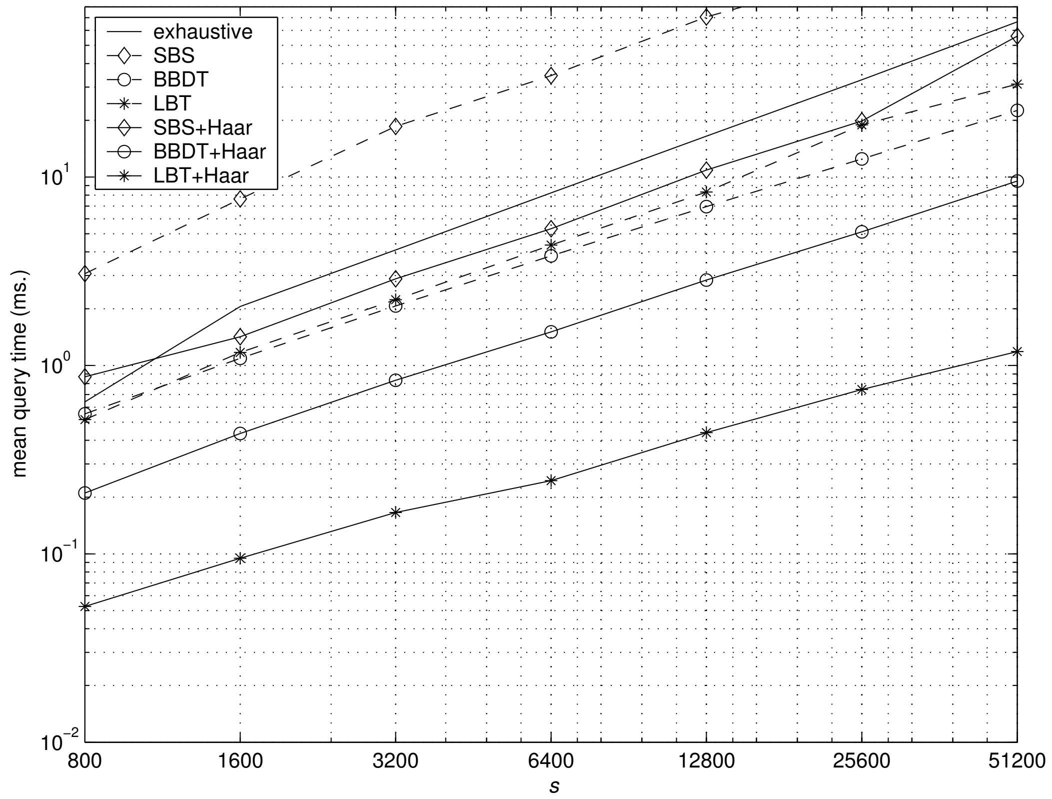
(a)

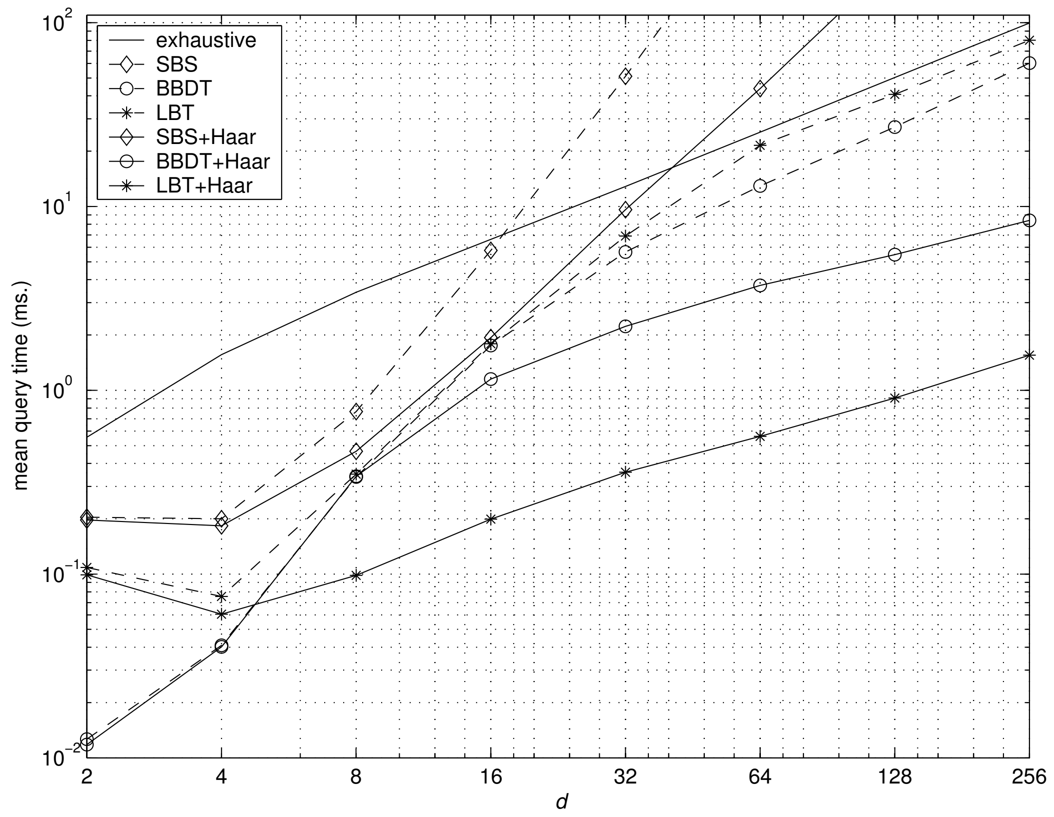


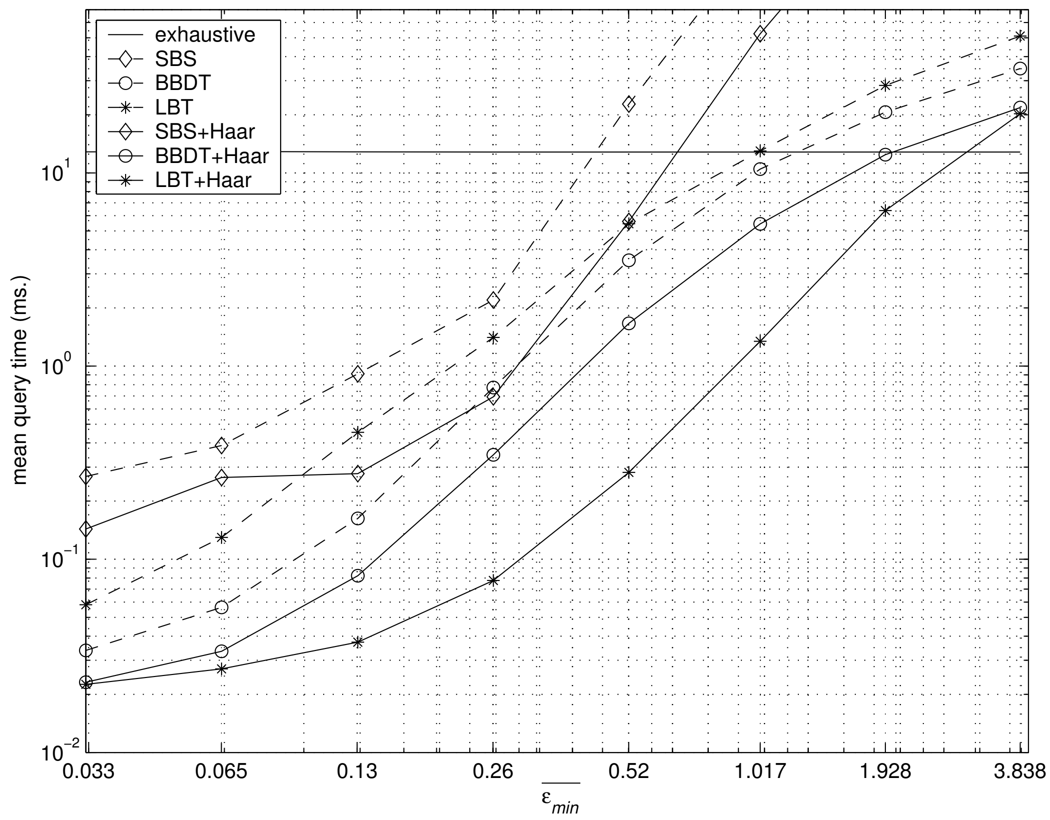
(b)



(c)


exhaustive
SBS
BBDT
LBT
SBS+Haar
BBDT+Haar
LBT+Haar
mean query time (ms.)
10^{1}
10^{0}
10^{-1}
10^{-2}
800
1600
3200
6400
12800
25600
51200
s


exhaustive
SBS
BBDT
LBT
SBS+Haar
BBDT+Haar
LBT+Haar
mean query time (ms.)
10^2
10^1
10^0
10^{-1}
10^{-2}
2
4
8
16
32
64
128
256
d


exhaustive
SBS
BBDT
LBT
SBS+Haar
BBDT+Haar
LBT+Haar
mean query time (ms.)
10^{1}
10^{0}
10^{-1}
10^{-2}
0.033
0.065
0.13
0.26
0.52
1.017
1.928
3.838
$\overline{\varepsilon_{min}}$