ELSEVIER

Information Processing Letters 81 (2002) 239–246

Information Processing Letters

www.elsevier.com/locate/ipl

An efficient nearest neighbor search in high-dimensional data spaces ☆

Dong-Ho Lee *, Hyoung-Joo Kim

Seoul National University, Department of Computer Engineering, San 56-1, Sillim-dong, Gwanak-gu, Seoul 151-742, Republic of Korea

Received 27 November 2000; received in revised form 21 May 2001
Communicated by K. Iwama

Keywords: Algorithms; Incremental nearest neighbor search; High-dimensional index structure; SPY-TEC

1. Introduction

Similarity search in multimedia databases requires an efficient support of nearest neighbor search on a large set of high-dimensional points. A technique applied for similarity search in multimedia databases is to transform important properties of the multimedia objects into points of a high-dimensional feature space. The feature space is usually indexed using a multidimensional index structure. Then, similarity search corresponds to a range search which returns all objects within a threshold level of similarity to the query objects, and a k-nearest neighbor search that returns the k most similar objects to the query object.

Initially, traditional multidimensional data structures (e.g., R-tree [1], kd-tree [5]), which were designed for indexing low-dimensional spatial data, were used for indexing high-dimensional feature vectors. However, recent research activities [10,9,8] reported the result that basically none of the querying and indexing techniques which provide good results on low-dimensional data also performs sufficiently well on high-dimensional data. Many researchers have called this problem the "*curse of dimensionality*" [3], and many database-related projects have tried to tackle it. As a result of these research efforts, a variety of new index structures [11,9], cost models [10] and query processing techniques [7] have been proposed. However, most of the high-dimensional index structures are extensions of the R-tree or the kd-tree adapted to the requirements of high-dimensional indexing. Thus, all of these index structures are limited with respect to data space partitioning and suffer from specific drawbacks of the R-tree or the kd-tree.

To overcome these drawbacks, in our earlier work we proposed the SPY-TEC [2], an efficient index structure for similarity search in high-dimensional spaces and proposed the algorithm for processing range queries on the SPY-TEC. However, we could not propose an algorithm for processing nearest neighbor queries on the SPY-TEC.

In this paper, we introduce a new metric that can be used to guide an ordered best-first traversal when finding nearest neighbors on the SPY-TEC. Based on

☆ This research was partially funded by the 1999 BK21 IT area grant of the Ministry of Education in Korea.

* Corresponding author.
E-mail addresses: dhlee@oopsla.snu.ac.kr (D.-H. Lee), hjk@oopsla.snu.ac.kr (H.-J. Kim).

PII: S0020-0190(01)00236-8

this new metric, we propose the incremental nearest neighbor algorithm on the SPY-TEC.

2. Related work

Roussopoulos et al. [7] proposed an algorithm for a nearest neighbor search in the R-tree. The key idea of their work is to maintain a global list (*ActiveBranchList*) of the candidate k nearest neighbors as the R-tree is traversed in a depth-first manner. The authors of [7] introduced two important distance functions, MINDIST and MINMAXDIST for ordering nodes that will be visited. MINDIST is the minimum distance from a query object q to a node (or bounding rectangle r) of the R-tree, while MINMAXDIST is the distance from q to the closest corner of r that is "adjacent" to the corner farthest from q. With these distance functions, the authors proposed three strategies for upward and downward pruning. In some sense, the two orderings represent the optimistic (MINDIST) and the pessimistic (MINMAXDIST) ordering choices because experiments reported in [7] showed that ordering the *ActiveBranchList* using MINDIST consistently performed better than using MINMAXDIST.

Hjaltason and Samet [4] proposed the incremental nearest neighbor algorithm that employs what may be termed best-first traversal. When finding k nearest neighbors to the query object using the algorithm proposed in [7], k is known prior to the invocation of the algorithm. Thus, if the $(k+1)$th neighbor is needed, the k-nearest neighbor algorithm needs to be reinvoked for $(k+1)$ neighbors from scratch. To resolve this problem, the authors of [4] proposed the concept of *distance browsing* which is to obtain the neighbors incrementally (i.e., one by one) as they are needed. They showed through various experiments that their incremental algorithm significantly outperforms the algorithm of [7] for distance browsing queries and also usually outperforms it when applied to the k-nearest neighbor problem for the R-tree. They also showed that, of the three pruning strategies proposed in [7], the one pruning strategy that does not use MINMAXDIST is sufficient when used in a combination of upward and downward pruning in their algorithm. This implies that MINMAXDIST is not necessary for pruning in the incremental nearest neighbor search.

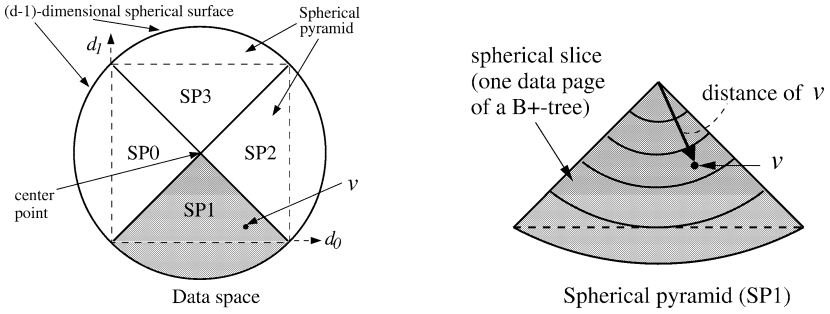
To the best of our knowledge, the incremental approach is one of the most efficient algorithms for finding the nearest neighbor or k nearest neighbors. However, this algorithm does not provide good results on high-dimensional data either, as we will show in our experimental evaluation. This is not a problem of the algorithm itself, but a problem of the underlying index structure (R-tree), which does not support efficient indexing or query processing structurally on a high-dimensional data space.

3. The SPY-TEC

In [9], Berchtold et al. proposed a special partitioning strategy, the Pyramid-Technique, which divides the d-dimensional data space first into $2d$ pyramids, and then cut the single pyramid into several slices. They also proposed the algorithm for processing hypercubic range queries on the space partitioned by this strategy. However, the shape of queries used in similarity search is not a hypercube, but a hypersphere [3]. Thus, when processing hyperspherical queries with the Pyramid-Technique, there is a drawback which exists in all index structures based on the bounding rectangle [3,2].

The main idea of the SPY-TEC is based on the observation that spherical splits will be better than right-angled splits of the Pyramid-Technique for similarity search. This observation is due to the fact that the shape of the queries used in similarity search is not a hypercube, but a hypersphere.

The SPY-TEC partitions the data space in two steps: In the first step, we split a d-dimensional data space into $2d$ spherical pyramids having the center point of the data space $(0.5, 0.5, \ldots, 0.5)$ as their top and a $(d-1)$-dimensional spherical surface of the data space as their bases. The second step is to divide each of the $2d$ spherical pyramids into several spherical slices with a single slice corresponding to one data page of the B^+-tree. We will call this spherical slice the "*bounding slice* (*BS*)" because it is a data page region. Fig. 1 shows the data space partitioning of the SPY-TEC in a 2-dimensional example. First, the 2-dimensional data space has been divided into 4 spherical pyramids. All of these spherical pyramids have the center point of the data space as their top and one spherical surface of the data space as their



(a) First step (b) Second step

Fig. 1. Partitioning strategy of the SPY-TEC.

bases. In the second step, each of these 4 spherical pyramids is split again into several data pages which are shaped like the annual ring of a tree. Although the formal expression of this procedure was presented in [2], we redefine it formally for better understanding of our incremental nearest neighbor algorithm on the SPY-TEC.

Definitions 1 and 2, which follow, correspond to the first step and second step of our partitioning strategy. That is, as depicted in Fig. 1, we can determine the spherical pyramid to which a point belongs, according to Definition 1. And, we can determine the location of a point in its spherical pyramid by Definition 2.

Definition 1 (*Spherical pyramid of a point* v). A d-dimensional point v is defined to be located in spherical pyramid sp_i.

$$i = \begin{cases} j_{\max} & \text{if } v_{j_{\max}} < 0.5, \\ (j_{\max} + d) & \text{if } v_{j_{\max}} \geqslant 0.5, \end{cases}$$

$$j_{\max} = \big(j \mid \big(\forall k,\ 0 \leqslant (j,k) < d,\ j \neq k: \\ |0.5 - v_j| \geqslant |0.5 - v_k|\big)\big).$$

Definition 2 (*Distance of a point* v). Given a d-dimensional point v, the distance d_v of the point v is defined as

$$d_v = \sqrt{\sum_{i=0}^{d-1} (0.5 - v_i)^2}.$$

Finally, Definition 3, which follows, is to transform a d-dimensional point v into a 1-dimensional value $(i \cdot \lceil \sqrt{d} \rceil + d_v)$ using Definitions 1 and 2.

Definition 3 (*Spherical pyramid value of a point* v). Given a d-dimensional point v, let sp_i be the spherical pyramid to which v belongs according to Definition 1, and d_v be the distance of v according to Definition 2. Then, the spherical pyramid value spv_v of v is defined as

$$spv_v = \big(i \cdot \lceil \sqrt{d} \rceil + d_v\big).$$

For example, consider a 2-dimensional point, $v = (0.4, 0.8)$. According to Definition 1, the point v belongs to $sp_{(1+2)}$ because $j_{\max}$ is 1 and v_1 $(= 0.8)$ is greater than 0.5. And, according to Definition 2, the distance from v to the center is $\sqrt{0.1}$. Finally, the spherical pyramid value (spv_v) of the point v is $3 \cdot \lceil \sqrt{2} \rceil + \sqrt{9}$ by Definition 3.

Using this partitioning strategy, the SPY-TEC can transform a d-dimensional data point into a one-dimensional value and then store a d-dimensional point *plus* the corresponding one-dimensional key as a record in the leaf nodes of a B^+-tree which provides fast insert, update, delete, and search operations. It is a very simple task to build an index using the SPY-TEC. Given a d-dimensional point v, we first determine the spherical pyramid value spv_v of the point and then insert the point into a B^+-tree using spv_v as a key. Finally, we store the point v and spv_v in the according data page of the B^+-tree. Update and delete operations can be done similarly.

4. Incremental nearest neighbor algorithm on the SPY-TEC

The algorithm proposed in [4] picks the node with the least distance in the set of all nodes that have yet to be visited when deciding what node to traverse next on the R-tree. This means that instead of using a stack or a plain queue to keep track of the nodes to be visited, it uses a priority queue where the distance from the query point is used as a key. In our algorithm, we also use a priority queue where the distance from the query point to the nodes or objects is used as a key.

4.1. Metrics for nearest neighbor search

For the incremental nearest neighbor search on the SPY-TEC, we need the minimum possible distance from the query object to a node in the SPY-TEC. Fig. 2 shows an example of the SPY-TEC in a two-dimensional data space. For the sake of simplicity, we assume that each bounding slice contains one object. In Fig. 2, the query point falls within a bounding slice BS_4 in the spherical pyramid sp_1. As with most nearest neighbor algorithms, we must first visit the page (BS_4 in this example) containing the query point. Then, we visit the next page with the second smallest minimum distance from the query point. To do so, we must calculate the minimum possible distance from the query point to a spherical pyramid or a bounding slice. We first describe the process of calculating the minimum distance between the query point and a spherical pyramid, and then discuss the process of calculating the minimum distance between the query point and a bounding slice.

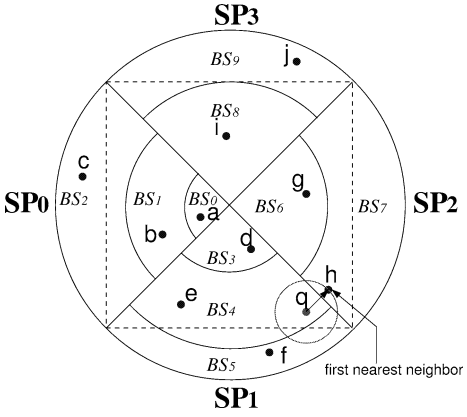


Fig. 2. An example of the SPY-TEC for a set of 10 points.

Lemma 1, which follows, measures the minimum distance $\text{MINDIST}(q, sp_i)$ from the query point q to a spherical pyramid sp_i. For the sake of simplicity, we focus on the description of the case only for spherical pyramids sp_i where $i < d$. However, this lemma can be extended to all spherical pyramids in a straightforward manner [2].

Lemma 1 (Minimum distance from a query point to a spherical pyramid). *Given a query point* ($q = [q_0, q_1, \ldots, q_{d-1}]$), *let* sp_j ($j < d$) *be the spherical pyramid containing a query point, and* sp_i *be the spherical pyramid that will be examined for the minimum possible distance from* q. *The minimum distance from* q *to* sp_i, $\text{MINDIST}(q, sp_i)$, *is defined as*

$$\text{MINDIST}(q, sp_i) = \begin{cases} 0 & \text{if } i = j, \\ d_q & \text{if } |i - j| = d, \\ \dfrac{|q_j - q_i|}{\sqrt{2}} & \text{if } i < d, \\ \dfrac{|q_j + q_i - 1|}{\sqrt{2}} & \text{if } i > d. \end{cases}$$

Proof. Due to lack of space, we only show each case by using a 2-dimensional example of Fig. 3 instead of the formal proof. First, the spherical pyramid sp_0 contains the query point q. Therefore, $\text{MINDIST}(q, sp_0) = 0$, which is less than or equal to the distance of q from any point in sp_0. And sp_1 is adjacent to q. Thus, the minimum distance of q from sp_1 is the length of the

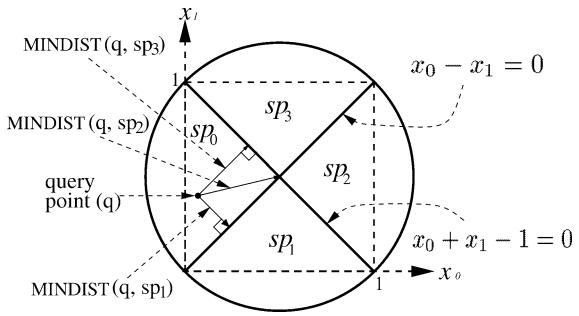
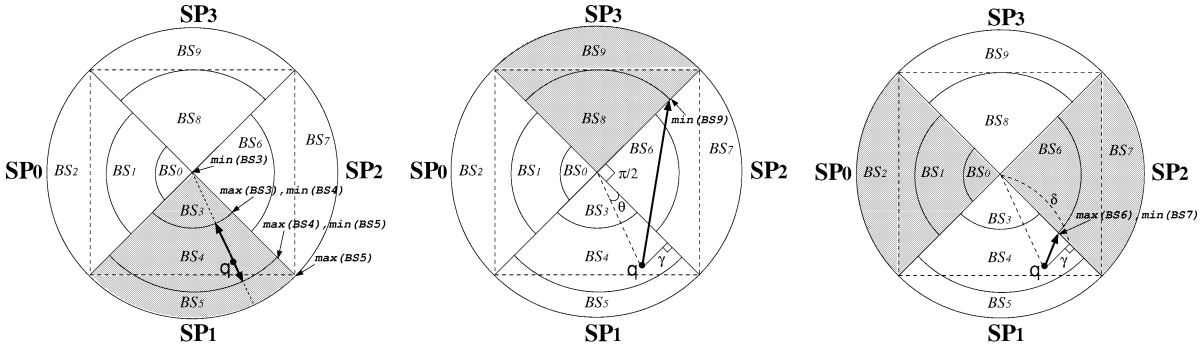


Fig. 3. The minimum distance from the query point to a spherical pyramid.



(a) Case 1 (b) Case 2 (c) Case 3

Fig. 4. The minimum distance from the query point to a bounding slice.

straight line (α) between q and the closest side plane of sp_1. Therefore,

$$\text{MINDIST}(q, sp_1) = |q_0 - q_1|/\sqrt{2}.$$

The rest of cases can be calculated analogously. For the formal proof, you can refer to [6]. □

Calculating the minimum distance from the query point to a bounding slice is more complex than the case of the minimum distance from the query point to a spherical pyramid. However, as depicted in Fig. 4 and Lemma 2, we can present it easily by classifying into three cases.

Lemma 2 (Minimum distance from a query point to a bounding slice). *Given a query point (q), let sp_j be the spherical pyramid containing a query point, and BS_l be the bounding slice that belongs to a spherical pyramid sp_i. The minimum distance from q to a bounding slice BS_l,* $\text{MINDIST}(q, BS_l)$*, is defined as*

Case 1: ($i = j$: *the case of BS_l belonging to the spherical pyramid that contains q.*)

$$\text{MINDIST}(q, BS_l) = \begin{cases} |d_q - \max(BS_l)| \\ \quad \text{if } d_q > \max(BS_l), \\ 0 \quad \text{if } \min(BS_l) \leqslant d_q \leqslant \max(BS_l), \\ |d_q - \min(BS_l)| \\ \quad \text{if } d_q < \min(BS_l). \end{cases}$$

Case 2: ($|i - j| = d$: *the case of BS_l belonging to the spherical pyramid on the opposite side of q.*)

Let α be the distance from the closest side plane of a spherical pyramid adjacent to q and θ ($\leqslant \pi/4$) be the angle of a right-angled triangle which consists of two sides, α and d_q ($\sin\theta = \alpha/d_q$),

$$\text{MINDIST}(q, BS_l) = \sqrt{{d_q}^2 + \min(BS_l)^2 - 2d_q \min(BS_l)\cos\left(\theta + \frac{\pi}{2}\right)}.$$

Case 3: (*otherwise*: *the case of BS_l belonging to a spherical pyramid adjacent to q.*)

Let δ be the length of the base line in a right-angled triangle which consists of two sides, α and d_q,

$$\text{MINDIST}(q, BS_l) = \begin{cases} \sqrt{|\delta - \max(BS_l)|^2 + \alpha^2} \\ \quad \text{if } \delta > \max(BS_l), \\ \alpha \quad \text{if } \min(BS_l) \leqslant \delta \leqslant \max(BS_l), \\ \sqrt{|\delta - \min(BS_l)|^2 + \alpha^2} \\ \quad \text{if } \delta < \min(BS_l), \end{cases}$$

where

$$\min(BS_l) = \{d_v \mid (\forall v',\ v, v' \in BS_l\colon\ d_v \leqslant d_{v'})\},$$

$$\max(BS_l) = \{d_v \mid (\forall v',\ v, v' \in BS_l\colon\ d_v \geqslant d_{v'})\}.$$

Proof. Due to lack of space, we only show Case 1 by using a 2-dimensional example of Fig. 4(a) instead

of the formal proof. In Fig. 4(a), BS_4 contains the query point q. Therefore, MINDIST(q, BS_4) is 0. And MINDIST(q, BS_3) is the difference between d_q and d_v, where the point v is in BS_3 and is farthest from the center of the space. Therefore,

$$\text{MINDIST}(q, BS_3) = \left|d_q - \max(BS_3)\right|.$$

Finally, MINDIST(q, BS_5) is the difference between d_q and d_v, where the point v is in BS_5 and is closest to the center. Therefore,

$$\text{MINDIST}(q, BS_5) = \left|d_q - \min(BS_5)\right|.$$

For the formal proof, you can refer to [6]. □

4.2. Algorithm description

Algorithm 1 shows the algorithm for processing the nearest neighbor query. In lines 1–4, the distances of each spherical pyramid from the query point are calculated by using Lemma 1, and then information about each spherical pyramid and its distance are inserted into the priority queue. Since the distance is used as a key in the priority queue, the spherical pyramid closest to the query point is at the head of the queue. The **while**-loop of lines 6–21 is the main loop for the algorithm. In line 7, the first element in the head of the queue is dequeued and, according to the type of the element, appropriate operations will be performed. If the type of the element dequeued is a spherical pyramid, as depicted in lines 8–12, the distances of each bounding slice in the spherical pyramid from the query point are calculated, and then information of each bounding slice and its distance are inserted into the queue by using Lemma 2. If the type is a bounding slice, as depicted in lines 13–17, the distances of each object in the bounding slice from the query point are calculated, and then inserted into the queue. Finally, if the type is an object, it is reported as the next nearest neighbor object. The first reported object is naturally the nearest neighbor to the query point. If we control the number of reported nearest neighbors in the **while**-loop of Algorithm 1, we can easily process the k-nearest neighbor query.

```
 1: for i = 0 to 2d − 1 do
 2:    dist = MINDIST(q, sp_i);    /* Using Lemma 1 */
 3:    ENQUEUE(queue, sp_i, dist);
 4: end for
 5:
 6: while not ISEMPTY(queue) do
 7:    Element = DEQUEUE(queue);
 8:    if Element is a spherical pyramid then
 9:       for each bounding slice in a spherical pyramid do
10:          dist = MINDIST(q, BS_l);    /* Using Lemma 2 */
11:          ENQUEUE(queue, BS_l, dist);
12:       end for
13:    else if Element is a bounding slice then
14:       for each object in a bounding slice do
15:          dist = DIST_QUERY_TO_OBJ(q, object);
16:          ENQUEUE(queue, object, dist);
17:       end for
18:    else    /* Element is an object */
19:       report element as the next nearest object
20:    end if
21: end while
```

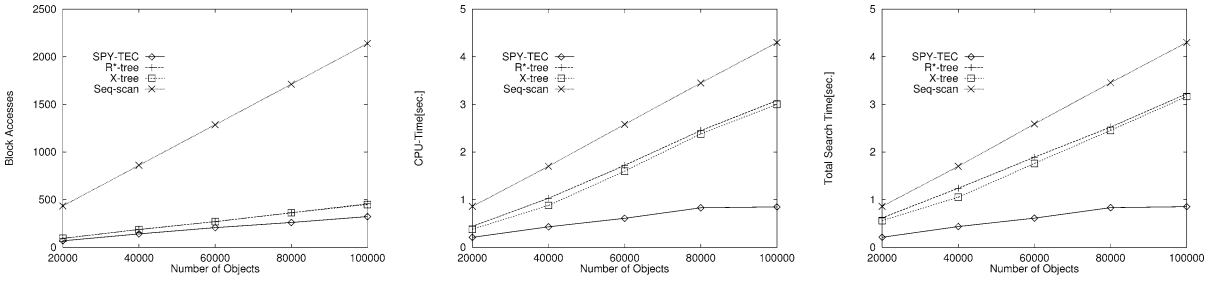
Algorithm 1. Processing the incremental nearest neighbor query.

4.3. Example

As an example, suppose that we want to find the first nearest neighbor to the query point q in the SPY-TEC given in Fig. 2. Below, we show the steps of the algorithm and the contents of the priority queue. Table 1 shows these distances (*SP* means spherical pyramid and *BS* means bounding slice). When depicting the contents of the priority queue, the spherical pyramids and bounding slices are listed with their distances from the query point q, in order of

Table 1
Distances of spherical pyramids and bounding slices from the query point q in the SPY-TEC of Fig. 2

SP	Dist.	*BS*	Dist.	OBJ	Dist.
SP_0	21	BS_0	21	a	23
SP_1	0	BS_1	25	b	27
SP_2	4	BS_2	29	c	45
SP_3	33	BS_3	14	d	16
		BS_4	0	e	19
		BS_5	2	f	12
		BS_6	8	g	35
		BS_7	4	h	6
		BS_8	33	i	39
		BS_9	42	j	47



(a) Block access (b) CPU time (c) Total search time

Fig. 5. Performance behavior on real data.

increasing distance. The objects are denoted in bold letters (e.g., **a**). The algorithm starts by enqueueing SP_0–SP_3, after which it executes the following steps:

1. Enqueue SP_0–SP_3. Queue: {[SP_1, 0], [SP_2, 4], [SP_0, 21], [SP_3, 33]}.
2. Dequeue SP_1, enqueue BS_3, BS_4, BS_5. Queue: {[BS_4, 0], [BS_5, 2], [SP_2, 4], [BS_3, 14], [SP_0, 21], [SP_3, 33]}.
3. Dequeue BS_4, enqueue **e**. Queue: {[BS_5, 2], [SP_2, 4], [BS_3, 14], [**e**, 19], [SP_0, 21], [SP_3, 33]}.
4. Dequeue BS_5, enqueue **f**. Queue: {[SP_2, 4], [**f**, 12], [BS_3, 14], [**e**, 19], [SP_0, 21], [SP_3, 33]}.
5. Dequeue SP_2, enqueue BS_6, BS_7. Queue: {[BS_7, 4], [BS_6, 8], [**f**, 12], [BS_3, 14], [**e**, 19], [SP_0, 21], [SP_3, 33]}.
6. Dequeue BS_7, enqueue **h**. Queue: {[**h**, 6], [BS_6, 8], [**f**, 12], [BS_3, 14], [**e**, 19], [SP_0, 21], [SP_3, 33]}.
7. Dequeue **h**, report **h** as the first nearest neighbor.

These operations are repeated until the user finds as many nearest neighbors as desired.

5. Experimental evaluation

We performed various experiments to show the practical impact of the incremental nearest neighbor algorithm on the SPY-TEC and compared it to the R*-tree and the X-tree, as well as the sequential scan.

For clear comparison, we implemented the incremental nearest neighbor algorithm on the R*-tree and the X-tree using the algorithm proposed in [4]. All experiments were performed on a SUN SPARC 20 workstation with 128 MByte main memory and 10 GByte secondary storage. The block size used for our experiments was 4 KBytes. Due to lack of space, we show only the experiment using real data sets, although we performed various experiments using synthetic data sets and real data sets. The real data consists of Fourier points [11] in 12-dimensional space. We performed 10-nearest neighbor queries with 100 query points that were selected from the real data itself, and varied the database size from 20,000 to 100,000.

Fig. 5 shows the result of the experiment using real data sets. In this experiment, the SPY-TEC, along with the R*-tree or the X-tree significantly outperform the sequential scan regardless of the database size. From this result, we found that the real data consists of well-formed clusters which are meaningful workloads for high-dimensional nearest neighbor queries. The speed-up of the SPY-TEC in the total search time ranges between 2.42 and 3.71 over the X-tree, between 2.85 and 3.78 over the R*-tree, and between 3.90 and 5.04 over the sequential scan. The performance behavior of the number of block accesses and of CPU time are analogous to that of the total search time. The index structures SPY-TEC, X-tree, and R*-tree significantly outperform the sequential scan in all cases, and the SPY-TEC also clearly yields a better performance than do the X-tree and the R*-tree.

6. Conclusions

In this paper, we proposed the incremental nearest neighbor algorithm on the SPY-TEC. We also introduced a new metric (MINDIST) that can be used to guide an ordered best-first traversal when finding nearest neighbors on the SPY-TEC.

In our future work, we plan to study the parallel version of the nearest neighbor algorithm on the SPY-TEC using an efficient declustering technique that distributes the data onto the disks so that the data which has to be read when executing a query are distributed as equally as possible among the disks.

References

[1] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: Proc. ACM SIGMOD Internat. Conf. on Management of Data, June 1984, pp. 47–57.
[2] D.H. Lee, H.J. Kim, SPY-TEC: An efficient indexing method for similarity search in high-dimensional data spaces, Data Knowledge Engrg. 34 (1) (2000) 77–97.
[3] C. Faloutsos, Fast searching by content in multimedia databases, Data Engrg. Bull. 18 (4) (1995).
[4] G.R. Hjaltason, H. Samet, Distance browsing in spatial databases, ACM Trans. Database Systems 24 (2) (1999) 265–318.
[5] J. Bentley, Multidimensional binary search trees used for associative searching, Comm. ACM 18 (9) (1975) 509–517.
[6] D.H. Lee, H.J. Kim, An efficient nearest neighbor search in high-dimensional data spaces, Seoul National University, CE Technical Report OOPSLA-TR1028, 2000, http://oopsla.snu.ac.kr/~dhlee/OOPSLA-TR1028.ps.
[7] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, in: Proc. ACM SIGMOD Internat. Conf. on Management of Data, 1995, pp. 71–79.
[8] K. Beyer, J. Goldstein, R. Ramakrishnan, U. Shaft, When is "Nearest Neighbor" meaningful?, in: Proc. 7th Internat. Conf. on Database Theory, January 1999, pp. 217–235.
[9] S. Berchtold, C. Böhm, H.-P. Kriegel, The pyramid-technique: Towards breaking the curse of dimensionality, in: Proc. ACM SIGMOD Internat. Conf. on Management of Data, 1998.
[10] S. Berchtold, C. Böhm, D.A. Keim, H.-P. Kriegel, A cost model for nearest neighbor search in high-dimensional data space, in: ACM PODS Symposium on Principles of Database Systems, Tucson, AZ, 1997.
[11] S. Berchtold, D.A. Keim, H.-P. Kriegel, The X-tree: An indexing structure for high-dimensional data, in: Proc. 22nd Internat. Conf. on Very Large Databases, September 1996, pp. 28–39.