A distance-scan algorithm for spatial access structures

Andreas Henrich

Praktische Informatik, Fachbereich Elektrotechnik und Informatik
Universität Siegen, D-57068 Siegen, Germany
Email: henrich@informatik.uni-siegen.de

Abstract

In geographic information systems it is often useful to select an object located closest to a given point or to scan the objects with respect to their distance to a given point in ascending order. An example for a query of this type would be to retrieve ten hotels with at least three stars lying closest to the venue of a conference. Various subtypes of similar queries exist. On the other hand, research in geometric access structures has concentrated mainly on range queries.

We present an efficient algorithm for closest- and distance-scan-queries of various kinds. Our algorithm is based on the nearest neighbour algorithm for k-d-trees given by Friedman et al. [FBF77] and refined by Sproull [Spr91]. We adapt this algorithm to external access structures and extend it to process a broader class of queries.

Furthermore we show that the algorithm can be applied to point objects as well as to non-point objects, and that it can be used with all spatial access structures using a hierarchical directory. Analytical and experimental results show the outstanding performance of the algorithm.

1 Introduction

In recent years, various efficient access structures for large sets of geometric objects have been developed. Most of these structures have been designed for multidimensional points [Rob81, NHS84, OMSD87, Fre87, LS89, SK90, KO91]. Some structures can also be used to maintain multidimensional intervals using the so-called transformation technique [Hin85, SK88]. Other structures have directly been designed to maintain non-point objects [Gut84, SRF87, BKSS90, HSW90, GB91].

The focus of most of the listed approaches has been on the performance of range queries. Nearest-neighbour-, closest- and distance-scan-queries have not been considered.

On the other hand, queries like the following are usual in geographic applications:

- Retrieve ten hotels with at least three stars lying closest to the venue of a conference.
- For a certain location sort all supermarkets within a radius of 20 kilometres in ascending order according to their distance to that location.

Obviously various similar queries exist. In most cases the target of such a query is not simply the nearest neighbour. Sometimes additional restrictions are made like in the first example, where the hotels should have at least three stars. Often more than one object is searched and also the need for a sorted result is typical.

In this paper we present an efficient algorithm which can be used to process all variants of closest- and distance-scan-queries. The algorithm is based on the nearest neighbour algorithm for k-d-trees given by Friedman et al. [FBF77] and refined by Sproull [Spr91]. We adapt this algorithm to external access structures and extend it to process not only nearest-neighbour-queries.

Our algorithm will be described in detail for the LSD-tree, but it can be used without any modification for all access structures using a k-d-tree as directory [Rob81, OMSD87, LS89, KO91]. Only minor modifications are needed for other access structures using a hierarchical directory. Furthermore we show that the algorithm can be applied to non-point objects as well. We present the modifications necessary to use the algorithm with non-point access structures based on the transformation technique. The algorithm can also be used with all other non-point access structures using a hierarchical directory, like e.g. the R-tree and its refinements [Gut84, SRF87, BKSS90].

Our paper is organised as follows: A more detailed discussion of the various subtypes of closest- and distance-scan-queries is given in section 2. Section 3 sketches the aspects of the LSD-tree which are relevant for the paper. The algorithm for point objects is presented in section 4. Section 5 deals with the extension to non-point objects. Analytical and experimental results are given in section 6 and section 7. Section 8 concludes the paper.

2 Closest- and distance-scan-queries

In this section we will point out in more detail what is meant by the terms *closest-query* and *distance-scan-query* (or *distance-scan* for short). With respect to the closest-query we distinguish two subcases:

1. In its basic form, a closest-query yields the object (or objects) stored in an access structure for which the geometric key attribute has minimal distance to a given point p.

Published in: Proceedings of the 2nd ACM Workshop on Advances in Geographic Information Systems, p. 136-143, Gaithersburg, Maryland, Dezember 1994 (ACM ISBN: 0-89791-750-2).

A typical example for a closest-query of this type would be to retrieve the hotel(s) with minimal distance from the venue of a conference.

2. Obviously a natural extension of the above query arises if we search the hotel(s) with minimal distance from the venue of a conference with at least three stars. Whereas under 1 only the geometric attribute of the objects is concerned, in this second case one or more further attributes are used to give an additional restriction.

Up to now we have considered closest-queries retrieving only the objects with minimal distance. In practice this will normally be too restrictive. If there is one hotel with three stars which costs \$150 at a distance of 10 miles and one hotel with three stars which costs \$80 at a distance of 11 miles, we would obviously like to become aware of the second hotel too. Hence, it would be interesting to have a list of all hotels sorted in ascending order according to their distance to the venue of the conference. This is exactly, what a distance-scan yields.

In most cases it will not be necessary to scan all objects stored in the access structure. There are three methods to restrict the number of objects to be scanned:

1. *We give an upper bound for the distance.*
2. *We give an upper bound for the number of objects in the result.*
3. *We tell the system interactively when we have seen enough.*

It might also be useful to combine a distance-scan and a range query, i.e. to sort only those objects lying in a given query region which may e.g. be a rectangle, a polygon, a sector or a ray. In figure 1 variant (a) shows a distance scan restricted to the objects lying in the shaded sector and variant (b) shows the restriction to a given ray, which might be useful for non-point objects e.g. in ray tracing applications.

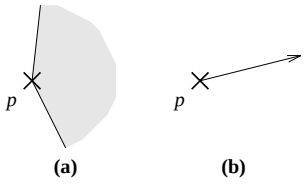

Figure 1: Combination of a distance-scan and a range query

All variants of closest-queries and distance-scans can be performed very efficient with the algorithm we will describe in section 4. Since we will explain this algorithm for the LSD-tree, we explain the basic concepts of the LSD-tree beforehand (*LSD* stands for *Local-Split-Decision*).

3 Basic concepts of the LSD-Tree

In the following we restrict our discussion to the two-dimensional case and describe only those aspects which are essential for this paper. For a detailed description we refer to [HSW89b, HSW89a].

For geometric data structures it is important to distinguish points and extended, i.e. non-point, objects. Conceptually the LSD-tree is a data structure for multi-dimensional points, but it was designed to maintain extended objects as well using the transformation technique, i.e. transforming k-dimensional axis-parallel rectangles into $2k$-dimensional points. In section 5 we will give a short description of the variant of the transformation technique we use together with the extension of the distance-scan algorithm for non-point objects, but in this section we restrict ourselves to points.

Geometric data structures which support spatial access to points usually divide the data space into pairwise disjoint data cells. With every data cell a bucket of fixed size is associated, which stores all objects contained in the cell. In this context a data cell is often called *bucket region*.

To introduce the basic idea of the LSD-tree we outline the construction of an LSD-tree, when new points are sequentially inserted.

Initially, the whole data space corresponds to one bucket. After a certain number of insertions the initial bucket has been filled, and an attempt to insert an additional object causes the need for a bucket split. To this purpose, a *split line* is determined and the objects on one side of the split line are stored in one bucket, while those on the other side are stored in another bucket. After some further insertions, the capacity of another bucket will be exceeded. In this case, a split line in the corresponding bucket region is determined, thus splitting this region into two subregions. This process is repeated each time the capacity of a bucket is exceeded.

For example, consider the two-dimensional data space in figure 2. The first split was made according to co-ordinate 50 in the first dimension, giving buckets 1 and 2. Then bucket 2 was split according to position 60 of the second dimension, giving buckets 2 and 3, and so on.

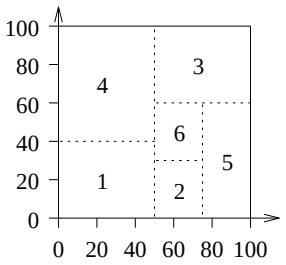

Figure 2: Possible data space partition for an LSD-tree

The split lines of the LSD-tree are maintained in a directory which is a generalised k-d-tree [Ben75]. For each split, a new node containing the position and the dimension of the split line is inserted into the directory tree. The leaves of the directory tree reference the buckets in which the actual objects are stored. For example, the partitioning displayed in figure 2 results in the directory tree displayed in figure 3.

Usually, the directory grows up to a point where it cannot be kept in main memory any longer. In this case, subtrees of the directory are stored in external pages on secondary memory, whereas the part of the directory near the root remains in main memory. For the details of the paging algorithm we refer to [HSW89b].

4 An algorithm for closest- and distance-scan-queries

In this section we will first of all introduce the algorithm for the basic distance-scan. Thereafter we explain how this algorithm can be used to process the other variants of the distance-scan and closest-queries.

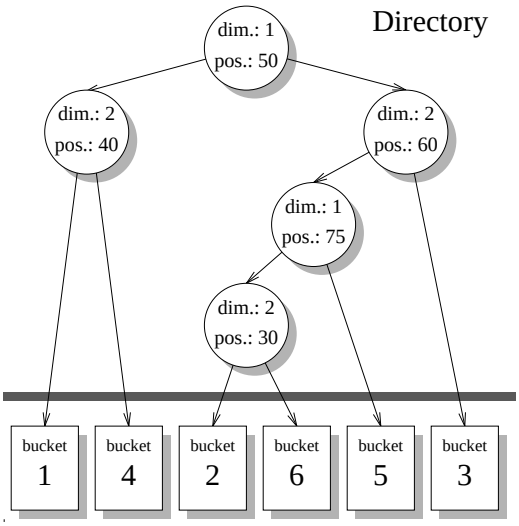

Figure 3: The LSD-tree associated with the data space partition of figure 2

As an introductory example we use the LSD-tree of figure 3. The objects stored in this LSD-tree are to be sorted according to their distance from the point $p = (60\,;\,35)$.

The algorithm starts searching the bucket for which the bucket region contains p. In our example this is bucket 6. This bucket is sparsely shaded in figure 4*i*. The objects stored in this bucket are inserted into an auxiliary data structure *OPQ*. *OPQ* is a priority queue in which the object with minimal distance to p has highest priority. Thereafter the minimal distance Δ_{min} between the point p and the closest edge of the sparsely shaded bucket region is computed.

Now objects are taken from the priority queue *OPQ* until the distance between the first element in *OPQ* and p is greater than Δ_{min}. In figure 4*i* the circular region containing the objects taken from *OPQ* up to now is densely shaded.

At this time the sparsely shaded region is extended with the bucket not yet shaded lying closest to p. In our example this is bucket 2. Hence, the objects stored in bucket 2 are inserted into *OPQ*. Δ_{min} is increased from 5 to 10 to reflect the extension of the sparsely shaded region and the objects with a distance to p less than or equal to 10 are taken from *OPQ*. Figure 4*ii* illustrates this step.

In the following the buckets are added to the sparsely shaded region in the sequence 1, 4, 5, 3. Each time a bucket is added, the objects stored in the bucket are inserted into *OPQ*, Δ_{min} is increased accordingly and the objects with a distance to p less than or equal to Δ_{min} are taken from *OPQ*. Figure 4 states the single steps.

During the described process, the auxiliary data structure *OPQ* always contains the objects lying in the union of the sparsely and the densely shaded regions. The densely shaded region indicates which objects can be taken from *OPQ* in the corresponding step in sorted order. The white circle around p shows the region for which the objects are already sorted (i.e. deleted from *OPQ*).

For a more detailed description of the algorithm we need an additional auxiliary data structure *NPQ* to store directory nodes and buckets for later examination. To this end an identifier for the directory node or bucket is stored in *NPQ* together with the corresponding *data region*. The *data region* of a bucket is its bucket region and the data region of a directory node is the union of the data regions of all buckets which can be reached from this directory node. Like *OPQ*, *NPQ* is a priority queue, and the directory node or bucket for which the data region has minimal distance to p has highest priority.

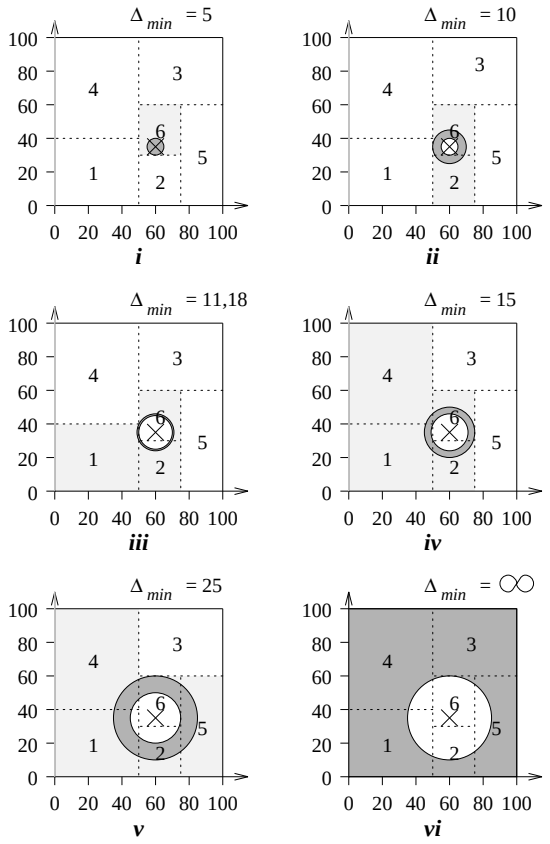

Figure 4: Example for the course of a distance-scan; $p = (60\,;\,35)$

With respect to the LSD-tree we can now describe the algorithm for a distance-scan as follows:

We start at the root of the directory and search for the bucket for which the bucket region contains p. Every time during this search when we follow the left son, we insert the right son into *NPQ* and every time we follow the right son, we insert the left son into *NPQ*.

Then the objects in the found bucket are inserted into *OPQ* and Δ_{min} is set. Thereafter the objects with a distance less than or equal to Δ_{min} are taken from *OPQ*.

Now the sparsely shaded region has to be extended. To this end the directory node or bucket with highest priority is taken from *NPQ*. If it happens to be a directory node, a new search is started choosing always the son on the side of the spilt line facing p. The other son is inserted into *NPQ*. The bucket determined in this way is processed inserting the objects stored in this bucket into *OPQ*, computing the new value for Δ_{min} and removing the objects with a distance less than or equal to Δ_{min} from *OPQ*. Thereafter the sparsely shaded region is extended in the same way again.

In order to state the distance-scan algorithm in pseudo-

code, we define the operations for the priority queues *OPQ* and *NPQ*. We assume that the priority queues can manage points as well as axis-parallel rectangles (i.e. data regions):

q := **CreatePQ**(p) creates a priority queue q in which the highest priority is given by the shortest distance to the point p.

PQInsert(q, o) inserts the object o into q.

o := **PQFirst**(q) assigns the object with highest priority in q to o.

PQDeleteFirst(q) deletes the object with highest priority from q.

IsEmptyPQ(q) checks if q is empty.

Figure 5 summarises the explained distance-scan algorithm. $D(w)$ denotes the data region of a directory node or bucket; w_{λ_i} denotes the lower bound of the data region of a directory node or bucket w in dimension i and w_{v_i} the upper bound. w_l and w_r are used to refer to the left, resp. right, son of a directory node w.

Obviously the algorithm given in figure 5 can be applied to all spatial access structures using a k-d-tree like directory. Furthermore the algorithm can be easily adapted for other access structures with a hierarchical directory. To this end, the computation of the data regions and the selection of the son to be followed, has to be changed accordingly. If the directory is not a binary tree, one way to perform the selection of the son would be to insert all sons into *NPQ* and then take the first element out of *NPQ* .

Now that we have described the basic algorithm for a distance-scan, it remains to show how this algorithm can be used to perform the other variants of the distance-scan and closest-queries.

- *basic closest-query:*
  When the first element is taken from *OPQ* its distance to p is stored as δ_{min}. The algorithm is stopped as soon as the distance of the first element in *OPQ* is greater than δ_{min}.
- *closest-query with an additional restriction:*
  All elements taken from *OPQ* not fulfilling the restriction are neglected. When the first element fulfilling the restriction is taken from *OPQ* its distance to p is stored as δ_{min}. The algorithm is stopped as soon as the distance of the first element in *OPQ* is greater than δ_{min}.
- *distance-scan with an upper bound ϕ for the distance:*
  The algorithm is stopped as soon as the distance of the first element in *OPQ* exceeds ϕ. Furthermore, buckets and directory nodes with a minimal distance between p and the corresponding data region greater than ϕ need not be inserted into *NPQ* and objects with a distance to p greater than ϕ need not be inserted into *OPQ* .
- *distance-scan with an upper bound χ for the number of the scanned objects:*
  The algorithm is stopped as soon as χ objects are taken from *OPQ*.
- *distance-scan combined with a range query:*
  Buckets or directory nodes for which the data region does not intersect the given query region, are not inserted into *NPQ* and objects lying outside the query region are not inserted into *OPQ*.

FUNCTION *DistanceScan* (p, T) **: QUEUE OF objects;**
{ yields as result a FIFO-queue containing all objects stored in the LSD-tree with directory T, sorted in ascending order according to their distance to point p }
BEGIN
  DistanceScan := EmptyQueue;
  CreatePQ(*NPQ*, p);
    { auxiliary data structure for the directory nodes and buckets to be examined later }
  CreatePQ(*OPQ*, p);
    { auxiliary data structure for the objects }
  w := root(T);
  $D(w) := (L_1; U_1] \times \ldots \times (L_k; U_k]$;
    { = the whole k-dimensional data space }
  PQInsert(*NPQ*, w);
  REPEAT
    w := PQFirst(*NPQ*);
    PQDeleteFirst(*NPQ*);
    WHILE w is not a bucket **DO**
      { w is a directory node → navigate to a bucket }
      s_{dim} := the split dimension stored in w;
      s_{pos} := the split position stored in w;
      $D(w_l) := (w_{\lambda_1}; w_{v_1}] \times \ldots \times (w_{\lambda_{s_{dim}}}; s_{pos}] \times \ldots \times (w_{\lambda_k}; w_{v_k}]$;
      $D(w_r) := (w_{\lambda_1}; w_{v_1}] \times \ldots \times (s_{pos}; w_{v_{s_{dim}}}] \times \ldots \times (w_{\lambda_k}; w_{v_k}]$;
      { stay on the side of the split line facing p }
      IF $p[s_{dim}] \leq s_{pos}$ **THEN** {*}
        { go left }
        PQInsert(*NPQ*, w_r);
        $w := w_l$;
      ELSE
        { go right }
        PQInsert(*NPQ*, w_l);
        $w := w_r$;
      END;
    END;
    { now w is a bucket }
    FOR EACH $o \in w$ **DO** PQInsert(*OPQ*, o) **END**;
    IF IsEmptyPQ(*NPQ*) **THEN**
      $\Delta_{min} := \infty$;
    ELSE
      $\Delta_{min} := \sqrt{\sum_{1 \leq i \leq k} (\max\{w_{\lambda_i} - p[i], p[i] - w_{v_i}, 0\})^2}$;
    END;
    REPEAT
      o := PQFirst(*OPQ*);
      IF dist(p, o) $\leq \Delta_{min}$ **THEN**
        QueueAppend(DistanceScan, o);
        PQDeleteFirst(*OPQ*);
      END;
    UNTIL dist(p, o) $> \Delta_{min}$ **OR** IsEmptyPQ(*OPQ*);
  UNTIL IsEmptyPQ(*NPQ*);
END *DistanceScan*;

Figure 5: Algorithm for a distance-scan

The given algorithm is obviously superior to traditional techniques. We will state some analytical and experimental performance results in section 6 and section 7.

5 The extension to non-point objects

We explain the non-point situation for k-dimensional intervals which serve as bounding boxes for arbitrary geometric objects in many applications.

To store a set of intervals in an LSD-tree we use the transformation technique, i.e. k-dimensional intervals are stored as $2k$-dimensional points. We choose the center representation which considers for each of the k dimensions the center and the half extension of the interval to be distinct dimensions. Figure 6 shows this transformation for $k = 1$. The 1-dimensional interval $[l_1; u_1]$ is represented by the 2-dimensional point $\left(\frac{u_1+l_1}{2}; \frac{u_1-l_1}{2}\right)$.

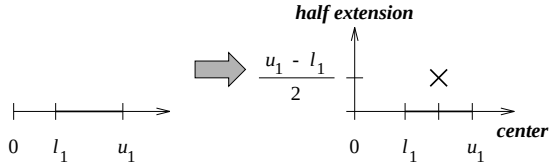

Figure 6: 1-dimensional example for the transformation technique (center representation)

The transformation technique has two pitfalls: (1) Using the transformation technique the data regions in the LSD-tree are $2k$-dimensional rectangles which can not be compared with k-dimensional query regions. (2) The transformation leads to a skew distribution of the image points.

With respect to our distance-scan algorithm especially the first aspect is of importance, because this algorithm is based on the existence of k-dimensional data regions for buckets and directory nodes geometrically including all k-dimensional objects stored in the corresponding part of the access structure.

Fortunately both pitfalls can be overcome.

The $2k$-dimensional data regions can be transformed into k-dimensional data regions. If the $2k$-dimensional data region of a directory node or bucket w is given by

$$D_{2k}(w) = (w_{\lambda_1}; w_{v_1}] \times \ldots \times (w_{\lambda_{2k}}; w_{v_{2k}}] \qquad (1)$$

a k-dimensional data region containing all rectangles for which the image point is in $D_{2k}(w)$ can be computed as follows:

$$\begin{aligned} D_k(w) = \quad & (w_{\lambda_1} - w_{v_2}; w_{v_1} + w_{v_2}] \times \ldots \\ & \ldots \times (w_{\lambda_{2k-1}} - w_{v_{2k}}; w_{v_{2k-1}} + w_{v_{2k}}] \end{aligned} \qquad (2)$$

Here the region for the centres is extended with the upper bound for the half extension. Figure 7 illustrates the relation between $D_{2k}(w)$ and $D_k(w)$. The data region $D_k(w)$ can be used for comparisons in the processing of range queries and distance-scans.

Experience shows that $D_k(w)$ tends to be much greater than the region covered by the intervals actually stored in the structure. Hence, we store the actual data region $\mathcal{D}_k(w)$, i.e. the minimal k-dimensional interval including all intervals stored in the corresponding part of the LSD-tree, with each

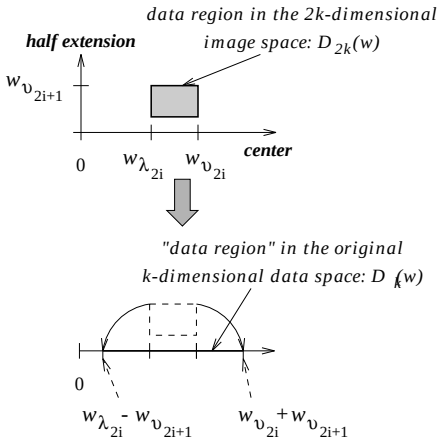

Figure 7: The relation between $D_{2k}(w)$ and $D_k(w)$

reference to a bucket or an external directory page in order to avoid unnecessary page accesses[1].

The second pitfall of the transformation technique, the skew distribution of the image points, can be handled by the flexible directory of an LSD-tree using a special split strategy:

- *split position:*
  If the split dimension is a dimension representing the centres of the intervals in one dimension, the data region is split in the middle. If the split dimension is a dimension representing the half extensions of the rectangles in one dimension, the mean of the half extensions of the objects stored in the bucket to be split is used as split position.

  The use of the mean for the split with respect to the extension brings up a great flexibility with respect to the type of objects to be stored. Even if the objects overlap in a high degree, this split strategy assures a well balanced directory.

- *split dimension:*
  The k dimensions of the original data space are used in cyclic order. Hence, on level i of the directory one of the dimensions $2\cdot(i \bmod k)$ or $2\cdot(i \bmod k)+1$ is used as split dimension. To chose among both dimensions, the split position is computed for both and the dimension with the smaller value for the sum of the areas of the resulting data regions $(\mathcal{D}_k(w_l) + \mathcal{D}_k(w_r))$ is chosen.

This split strategy has proven to be extremely flexible. It works well even for very skew object distributions and for object sets with strongly overlapping objects.

Now we are in the position to consider the modifications necessary to use the algorithm given in figure 5 for non-point objects. Only three changes are needed:

1. In the *if* -statement indicated by {*} in figure 5 we go left if the minimal distance between $\mathcal{D}_k(w_l)$ and p is less than the minimal distance between $\mathcal{D}_k(w_r)$ and p. Otherwise we go right[2]. It has to be noted, that

[1] Actually we do not store $\mathcal{D}_k(w)$ itself but a coding of $\mathcal{D}_k(w)$ relative to $D_k(w)$ in order to save space in the directory.

[2] To be more precise, we use $\mathcal{D}_k(w_l)$ and $\mathcal{D}_k(w_r)$ if we deal with references to buckets or external directory pages. Otherwise we use $D_k(w_l)$ and $D_k(w_r)$.

the distance can be zero for w_l and w_r. In this case a random choice can be made because in the following zero will be computed as the value of Δ_{min} and hence no element will be taken from OPQ until the objects in the other son have been examined.

2. In OPQ the objects have to be sorted in the same way as they are to be sorted in the result of the distance-scan. Normally this will mean that complex k-dimensional geometric objects approximated by k-dimensional bounding intervals must not be sorted in accordance to the minimal distance between p and the bounding interval but in accordance to the minimal distance between p and the complex object itself.

3. The computation of Δ_{min} has to be modified. The simplest method is now to take the distance between p and the first element in NPQ. To this end, the actual data region $\mathcal{D}_k(w)$ is stored with each reference to a directory node or bucket w in NPQ.

Obviously these are minor changes and the experimental results given in section 7 show that the algorithm works well for point and non-point objects. Furthermore, the algorithm can be adapted to other non-point access structures as described for the point situation.

6 Analytical considerations

In this section we sketch first analytical results with respect to the number of elements in the auxiliary data structure OPQ. We assume that N uniformly distributed points are stored in the access structure. The bucket capacity is b and the bucket utilisation is about $\ln 2$, the usual value for point access structures[3].

To simplify the situation further, we assume that the bucket regions build a grid over the data space, i.e. all bucket regions are quadratic and of equal size. We assume a 2-dimensional data space $[0\,;1]\times[0\,;1]$. Then the area of each bucket region is $\frac{b\cdot\ln 2}{N}$ and the length α of the edges of the bucket regions is given by

$$\alpha = \sqrt{\frac{b\cdot\ln 2}{N}}. \tag{3}$$

If we assume uniformly distributed objects, the area of the circular region covered by a distance-scan after scanning n objects will be $\frac{n}{N}$. From $\pi\cdot r^2 = \frac{n}{N}$ we can calculate the radius r of the scanned region:

$$r = \sqrt{\frac{n}{N\cdot\pi}} \tag{4}$$

Only objects located in buckets for which the bucket region intersects the edge of the scanned region are stored in the auxiliary data structure OPQ. Hence, we are interested in the number of bucket regions intersected by the edge of the scanned region. Figure 8 illustrates that under our assumptions the number of bucket regions intersected by the edge of a circle is exactly the number of bucket regions intersected by the minimal square including the circle. (The proof is omitted here because of space limitations.)

Each edge of a square with edge length $2\cdot r$ intersects at most $\left\lfloor\frac{2\cdot r}{\alpha}\right\rfloor+2$ bucket regions. Each bucket region containing

[3] In [HSW89a] an analysis shows this value as average bucket utilisation for the LSD-tree under fair assumptions and experimental results yield the same value.

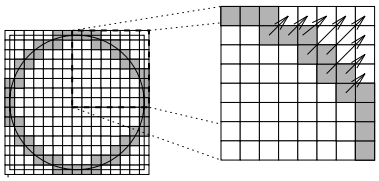

Figure 8: The bucket regions intersected by a circular region and its minimal including square

a corner of the square is intersected by two edges; hence, we can subtract 4 buckets. Then we can give the following bound for β the number of intersected bucket regions:

$$\beta \le 4\cdot\left(\left\lfloor\frac{2\cdot r}{\alpha}\right\rfloor+1\right) \tag{5}$$

From the objects stored in these buckets only those located outside the scanned region are still in OPQ. For $n \gg b$ on the average half of the area of each intersected bucket region will be inside, resp. outside, the scanned region. Hence, on the average $\frac{b\cdot\ln 2}{2}$ objects will be stored in OPQ for each intersected bucket and we get the following estimation for κ, the number of objects stored in OPQ after scanning n objects:

$$\kappa \le \frac{b\cdot\ln 2}{2}\cdot 4\cdot\left(\left\lfloor\frac{2\cdot r}{\alpha}\right\rfloor+1\right) \tag{6}$$

If we substitute the results of 3 and 4 in 6, we get:

$$\begin{aligned}\kappa &\le \tfrac{b\cdot\ln 2}{2}\cdot 4\cdot\left(\left\lfloor\frac{2\cdot\sqrt{\frac{n}{N\cdot\pi}}}{\sqrt{\frac{b\cdot\ln 2}{N}}}\right\rfloor+1\right)\\ &\le 1.88\cdot\sqrt{b\cdot n}+b\cdot 1.39\end{aligned} \tag{7}$$

Using this result we can give a rough estimation of the average case complexity of the algorithm given in section 4 for $n \gg 0$. Three operations dominate the time complexity of the algorithm:

1. The operations on NPQ: During a complete distance-scan the number of directory nodes and buckets inserted in and deleted from NPQ is exactly the number of buckets in the access structure minus one.

2. Traversing the directory: With a complete distance-scan each reference in the directory pointing from one directory node to a bucket or another directory node is traversed exactly once in the WHILE-loop in figure 5.

3. The operations on OPQ: During a complete distance-scan each object is inserted in and deleted from OPQ exactly once.

If we take into account, that for usual bucket capacities, say $b \ge 5$, the number of buckets as well as the number of pointers in the directory are by far smaller than the number of objects in the access structure, we can restrict our estimation to the objects inserted into OPQ.

If we use a heap data structure for example, the insertion can be done in $O(\log m)$ if OPQ contains m elements. The deletion can be done in $O(1)$.

If we sort n objects using a distance scan, we have to insert these n objects plus the objects stored in OPQ after sorting these n objects. According to our above considerations, we get $n + 1,88 \cdot \sqrt{b \cdot n} + b \cdot 1.39$ examined objects. The dominant operation for each object is the insertion into OPQ. Hence, we get the following effort:

$$\left(n + 1,88 \cdot \sqrt{b \cdot n} + b \cdot 1.39\right) \cdot O(\log(1,88 \cdot \sqrt{b \cdot n} + b \cdot 1.39)) \quad (8)$$

Therefore, for $k = 2$ we get a time complexity of $O(n \cdot \ln(\sqrt{n}))$ for the average case. The same considerations can be made for $k = 3$ and yield $O(n \cdot \ln(n^{\frac{2}{3}}))$ and in general we get $O(n \cdot \ln(n^{\frac{k-1}{k}}))$.

7 Experimental results

In this section we present experimental results for our distance-scan algorithm for point and for non-point objects.

Table 1 and table 2 state the results for point objects. We inserted 100,000 uniformly distributed points into an LSD-tree with bucket capacity 10. The bucket utilisation has been 68.9 %. p has been randomly chosen to be (0.108 ; 0.587) within the data space $[0\ ;1] \times [0\ ;1]$. In addition, table 1 contains a column with the upper bound for the expected number of objects in OPQ calculated according formula (7). There are two main reasons why the experimental values are below the expected values: (1) Formula (7) states an upper bound and (2) for $n > 4,096$ the edge of the scanned region is partly outside the data space. Nevertheless, the values for $n \leq 4,096$ show a satisfactory correlation.

It should be noticed, that we had to examine only 351 (= 256+95) objects at all in order to scan 256 objects. Only 51 bucket and 7 directory page accesses have been necessary. If we take into account, that 51 buckets could hold only 510 objects in the best case, this shows the excellent performance of the presented algorithm.

n = number of scanned objects	expected no. of objects in OPQ	max. no. of objects in OPQ	max. no. of objects in NPQ
1	20	9	15
16	38	22	17
256	109	95	37
4,096	395	332	104
16,384	775	488	153
65,536	1537	704	216
100,000	1895	704	216

Table 1: Experimental results for point objects

Table 3 states the results for non-point objects maintained in an LSD-tree using the refined transformation technique described in section 5. 100,000 uniformly distributed rectangles have been inserted into an LSD-tree with bucket capacity 10. The bucket utilisation has been 68.8 %. p is again (0.108 ; 0.587). The sum of the areas of all rectangles is 2.5-times the area of the data space.

If we compare the results for point and non-point objects, we can see that the performance of our algorithm is nearly as good for non-point objects.

The similarity between the values for point and non-point objects is surprising, if we take in to account, that the rectangles are stored as 4-dimensional points and that

n = number of scanned objects	bucket accesses	directory page accesses
1	1	2
16	4	2
256	51	7
4,096	633	58
16,384	2,440	186
65,536	9,564	659
100,000	14,516	973

Table 2: Experimental results for point objects

n = number of scanned objects	max. no. of objects in OPQ	max. no. of objects in NPQ	bucket accesses	directory page accesses
1	27	26	3	3
16	35	38	6	5
256	116	58	52	9
4,096	363	170	639	58
16,384	581	268	2,456	190
65,536	813	374	9,578	657
100,000	813	374	14,534	963

Table 3: Experimental results for non-point objects

our analytical considerations show, that the performance of the algorithm is worse for higher dimensions. To clarify the situation we have stated values for 1-, 2- and 3-dimensional intervals in table 4 and table 5. The results show the influence of the number of dimensions. On the other hand it comes out clearly, that the performance of the algorithm for k-dimensional intervals corresponds to the performance for k-dimensional points and not to the performance for $2k$-dimensional points one might have expected since we use the transformation technique. By the way this shows the quality of the used variant of the transformation technique.

n = number of scanned objects	maximum number of objects in OPQ		
	$k = 1$	$k = 2$	$k = 3$
1	10	27	10
16	15	35	108
256	17	116	425
4,096	23	363	1,870
16,384	29	581	3,515
65,536	29	813	4,990
100,000	29	813	4,990

Table 4: Experimental results for 1-, 2- and 3-dimensional intervals

Although the presented results are achieved with uniformly distributed objects, skew object distributions hardly deteriorate the performance of the algorithm.

8 Conclusion

We have presented an algorithm for closest-queries and distance-scans. The algorithm can be used for various subtypes of similar queries as well. Furthermore it can be used for point and non-point access structures as long as they use a hierarchical directory.

n = number of scanned objects	maximum number of objects in NPQ		
	$k=1$	$k=2$	$k=3$
1	17	26	42
16	17	38	73
256	17	58	206
4,096	22	170	737
16,384	26	268	1,164
65,536	26	374	1,545
100,000	26	374	1,545

Table 5: Experimental results for 1-, 2- and 3-dimensional intervals

The performance of the algorithm has been shown by analytical considerations and experimental results to be excellent. The algorithm is implemented in the Gral-System [Güt89] and has proved to be extremely useful.

In the near future we will develop the algorithm in two directions: We want to evaluate fields of application for the combination of the distance-scan and range queries. Computer animation, ray tracing or measurement evaluation may be interesting fields in this respect. On the other hand we try to use the algorithm for information retrieval problems. Here documents can be represented as points in a k-dimensional space, each dimension standing for the relevance with respect to a certain term. A query states a point in the k-dimensional space and searches the documents near this point since they are thought to have high relevance w.r.t. the query. The problem in this case is the number of dimensions ($k \gg 0$). On the other hand there is a strong correlation between the values in the single dimensions and we assume the correlation can be exploited using a sophisticated split strategy.

References

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, Atlantic City, 1990.

[FBF77] J.H. Friedman, J.L. Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Software*, 3:209–226, September 1977.

[Fre87] M. Freeston. The BANG file: a new kind of grid file. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pages 260–269, San Francisco, 1987.

[GB91] O. Günther and J. Bilmes. Tree-based access methods for spatial databases: implementation and performance evaluation. *IEEE Trans. on Knowledge and Data Eng.*, pages 342–356, 1991.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, Boston, 1984.

[Güt89] R.H. Güting. Gral: An Extensible Relational Database System for Geometric Applications. In *Proc. 15th International Conference on Very Large Data Bases*, pages 33–44, Amsterdam, 1989.

[Hin85] K. Hinrichs. *The grid file system: implementation and case studies of applications*. PhD thesis, ETH Zürich, 1985. Dissertation Nr. 7734.

[HSW89a] A. Henrich, H.-W. Six, and P. Widmayer. Paging binary trees with external balancing. In *Proc. 15th International Conference on Graph-Theoretic Concepts in Computer Science (WG'89)*, pages 260–276, Aachen, 1989.

[HSW89b] A. Henrich, H.-W. Six, and P. Widmayer. The LSD tree: spatial access to multidimensional point and non point objects. In *Proc. 15th International Conference on Very Large Data Bases*, pages 45–53, Amsterdam, 1989.

[HSW90] A. Hutflesz, H.-W. Six, and P. Widmayer. The R-File: An Efficient Access Structure for Proximity Queries. In *Proc. IEEE 6th Int. Conf. on Data Engineering*, pages 372–379, 1990.

[KO91] M.J. van Kreveld and M.H. Overmars. Divided k-d Trees. *Algorithmica*, 6:840–858, 1991.

[LS89] D.B. Lomet and B. Salzberg. A Robust Multi-Attribute Search Structure. In *Proc. IEEE 5th Int. Conf. on Data Engineering*, pages 296–304, 1989.

[NHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The Grid File: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[OMSD87] B.C. Ooi, K.J. McDonell, and R. Sacks-Davis. Spatial kd-Tree: An Indexing Mechanism for Spatial Databases. *IEEE COMPSAC*, pages 433–438, 1987.

[Rob81] J.T. Robinson. The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pages 10–18, 1981.

[SK88] B. Seeger and H.-P. Kriegel. Techniques for design and implementation of efficient spatial access methods. In *Proc. 14th International Conference on Very Large Data Bases*, pages 360–371, 1988.

[SK90] B. Seeger and H.-P. Kriegel. The buddy-tree: an efficient and robust access method for spatial data base systems. In *Proc. 16th International Conference on Very Large Data Bases*, pages 590–601, Brisbane, 1990.

[Spr91] R.F. Sproull. Refinements to Nearest-Neighbor Searching in k-Dimensional Trees. *Algorithmica*, 6:579–589, 1991.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: a dynamic index for multi-dimensional objects. In *Proc. 13th International Conference on Very Large Data Bases*, pages 507–518, 1987.