H. Tropf, H. Herzog: "Multidimensional Range Search in Dynamically Balanced Trees". Angewandte Informatik (Applied Informatics) 2/1981, pp 71..77. Vieweg Verlag, Wiesbaden, Germany.

Multidimensional Range Search in Dynamically Balanced Trees

H. Tropf, H. Herzog,

Fraunhofer-Institut für Informations- und Datenverarbeitung IITB,
Sebastian-Kneipp-Straße 12/14, 7500 Karlsruhe 1

Key-words: *data structures; range query; range search, multidimensional searching; search trees; balanced trees*

Abstract: *A mapping from multidimensional data (multi-key records) to one dimension is described. It simply consists of bitwise interlacing the keys. This mapping allows to use dynamically balanced binary search trees for efficient multidimensional range searching. A range search algorithm for bitwise interlaced keys is presented. Experimental results show that for small hypercube ranges the average number of records to be inspected is logarithmic with the number of records. Storage requirements are only two pointers per record to establish the search tree.*

Stichworte: *Datenstrukturen, Bereichsanfrage, mehrdimensionales Suchen, sortierte Bäume, ausgeglichene Bäume*

Zusammenfassung: *Es wird eine Abbildung mehrdimensionaler Daten (Records mit mehreren Schlüsseln) auf eine Dimension beschrieben. Die Abbildung erfolgt einfach durch bitweises Verzahnen der Schlüssel. Diese Art der Abbildung erlaubt eine effiziente Behandlung mehrdimensionaler Bereichsanfragen. Dafür wird ein Suchalgorithmus vorgestellt. In Experimenten wurde festgestellt, daß für kleine, gleich lange Schlüsselbereiche die durchschnittliche Anzahl der zu inspizierenden Records logarithmisch mit der Gesamtzahl der Records steigt. Es werden nur zwei Zeiger pro Record für die Baumstruktur benötigt.*

1. Introduction

Range searching is one of the standard requirements of database systems: each record having keys within some specified range is to be reported. In a distributing house database, e.g. all customers with age between 20 and 30 an postal number between 7500 and 7599 may be queried. Quite another field where range queries must be performed is pattern recognition (where the work described here was triggered off). Here, near neighbour searching is a subproblem that can be treated as range searching. Image analysis is a subfield of pattern recognition where there is typically the following situation: After preprocessing the raw image data, one has to deal with a sometimes tremendous amount of pattern primitives each given by several properties. For example, there are edge elements given by position (x and y value), length, and orientation which corresponds to a 4-key-record for each edge element. In the course of the anylysis, range searching in this data volume is a standard procedure. When dealing with real image analysis applications, algorithms must be fast; so it is worth worrying about how to perform range searches efficiently. The more advanced pattern recognition systems have a feedback from analysis to preprocessing which results in dynamic volumes of pattern primitives to be handled.

N records of k keys can be regarded as N points in a k dimensional space. In [1] a survey is given on problems that arise and data structures that are in use for the $k > 1$ case. A review of data structures for range searching, especially, is given in [2]. One of the basic structures is the k-d ("k-dimensional") tree. In [3] it is discussed in the general database context. In [4] a general technique, called "multidimensional divide an conquer", is described that leads to a data structure called "range tree". It has $O(N \log^{k-1} N)$ storage requirement and $O(\log^k N+F)$ query time to search the structure, where F is the number of records found.

The k-d tree has bee shown to be well adapted to range searching. Unfortunately, there is one disadvantage: there is no technique known to rebalance the trees when inserting or deleting a record. In this paper, a technique is proposed appearently sharing the advantageous properties of k-d trees for range searching, which allows to use any standard tree balancing mechanism.

It simply consists of bitwise interlacing the keys to form a one dimensional code. Because this is a mapping from k dimensions to one dimension, any tree balancing mechanism can be used for inserting and deleting. For range searching, an algorithm for bitwise interlaced data is presented. It turns out to be simply realized by basic bit manipulations. Experimental results indicate logarithmic time complexity: for small hypercube ranges the average number of records to be inspected is logarithmic with the number of records.

2 Bitwise Interlacing the Keys for Range Searching

In this chapter, bitwise interlacing is inductive introduced. After that, the relationship to k-d trees is discussed informally.

The goal is to find a bijective mapping from multi-key records to one dimension so that multidimensional range queries can be efficiently performed. Multi-key records can be considered as points in a multidimensional space. At first, we treat the two dimensional case where each point is given by its x and y value in the plane. The generalization to more dimensions, after that, turns out to be straightforward.

Given k keys of l bit length, the one dimensional code must have k*l lenght when the mapping is bijective. A naive approach would be to concatenate the keys to form a code as depicted in Fig. 1. This, of course, is a bijective mapping. It leads to sorting primarily according to x value; points with identical x value are sorted according to y value. After sorting, an exact match query is performed in O (k*log N) time, N being the number of records. Insertion and deletion are performed in O (k*log N) time, provided that dynamically balanced trees are chosen as data structure. However, it is not the right choice for range searches, as illustrated in Fig. 2: Although only a few or even no points may be within the search range, it is expected that many points (having x values within the specified range) must be inspected.

With the example of range searching in a geographical data base in mind, Knuth wrote in 1973 [5]: "Perhaps the best approach is to partition the set of all possible LATITUDE and LONGITUDE values rather coarsely ..., then to have on inverted list for each combined (LATITUDE, LONGITUDE) class". This partition can be done by dividing the plane into coarse slices after x value, which in turn are divided after y value. This is illustrated in Fig. 3, where each direction is divided up into ten regions. Here, when range searching, we must only inspect the points lying in the squares overlapping the search range (see Fig. 3). The coarse squares, in turn, can recursively be subdivided as shown in Fig. 4, for further reducing the number of points to be inspected.

How can this type of partition be referred to mapping? Looking at Figs. 3 and 4 (where we have chosen ten subregions for each division) one recognizes that it is equivalent to a mapping which is simply realized by interlacing the digits of the keys: Firstly we exclusively worry about the first digit of x and y, after that we consider the following digits. Interlacing the digits is an improved mapping from multidimensional data to one dimension for efficient range searching.

How can the mapping be further improved? The answer is simply to take the bits instead of the digits driving the idea of stepwise refinement and alternating x and y values to the extreme. Bitwise interlacing the keys is the simple mapping scheme we shall discuss here. It is illustrated in Fig. 5 and is easily expanded to more dimensions in the obvious way.

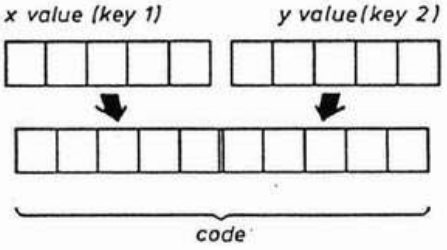


Fig. 1 A naive approach: Concatenating the keys to form a one dimensional code

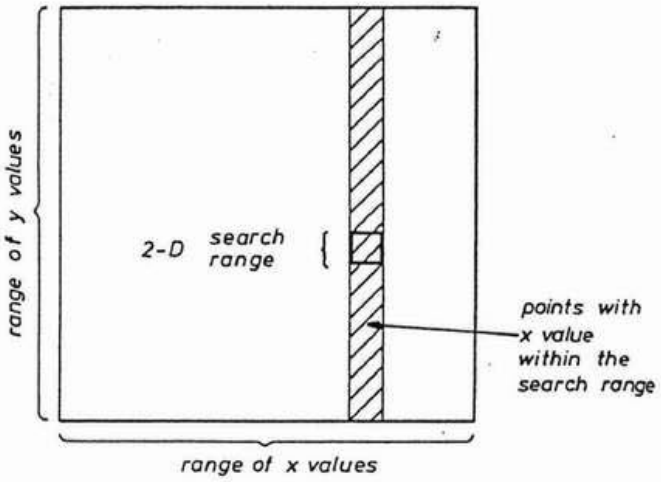


Fig. 2 Range searching when coding after Fig. 1

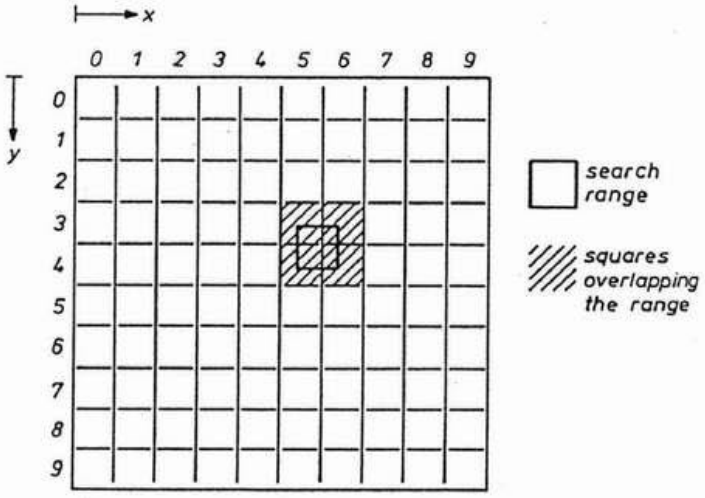


Fig. 3 Dividing the plane into coarse squares. Here, only the squares overlapping the search range must be inspected

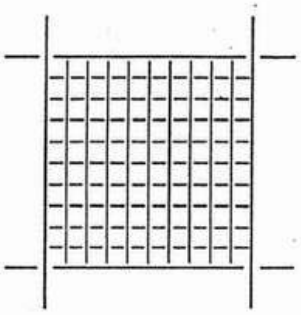


Fig. 4
Recursively subdividing the squares

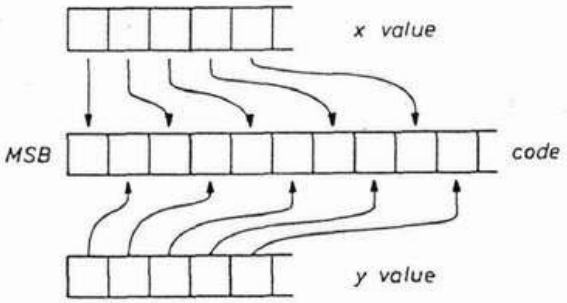


Fig. 5 Bitwise interlacing the keys (x and y value) to form the code

In Fig 6 the codes are depicted for the two dimensional case where the x and y values range from 0 to 7. One easily recognizes the recursive coding scheme.

Let us look at the bitwise interlacing scheme from some other point of view: One way of thinking of it is that nearby multidimensional data points should also lie in some close neighbourhood after mapping to one dimension. Of course, a bijective mapping from multidimensional data to one dimension cannot be done the way that in any case nearby multidimensional points are also close together in one dimension. Nevertheless, the coding scheme shown here tries to group them together. This "grouping feature" is desirable when searching for compact ranges. In this sense, there is some relationship to k-d trees which have been shown to be well suited for range searching: They also group the data points together alternating the discriminating keys.

The data structuring scheme of k-d trees is illustrated in Fig. 7: Point 1 corresponds to the root. The points the x values of which are greater than the x values of the root are in the right subtree of the root (the root of this subtree, node 2, corresponds to point 2). All other points are in the left subtree. The points of the right subtree the y values of which are greater than the y values of point 2 are in the right subtree of node 2, etc. Each node of a k-d tree has a discriminator which indicates the discriminating dimension. This discriminator may be implicitly given by reducing the level of the node modulo k (k = dimension). The essential difference between the k-d trees and bitwise interlacing the keys it as follows: *in k-d trees the discriminating dimension is a feature of the data structuring scheme. Bitwise interlacing, on the other hand, is a feature of the data representation itself. So we have still the free choice of structuring the data. This freedom leads to easy restructuring when handling dynamic data volumes.*

Two advantageous characteristics are shared by the k-d trees: (1) x and y values are almost equal in importance (of course one of them must be the first; in Fig. 5 the x value is the more important one); (2) the scheme is easily extended to more dimensions. It is easy to see that for equally spaced data points sorting the bitwise interlaced codes is equivalent to the k-d tree structuring mechanism.

An advantage of bitwise interlaced data over k-d trees is that the data structure adapts automatically to the data

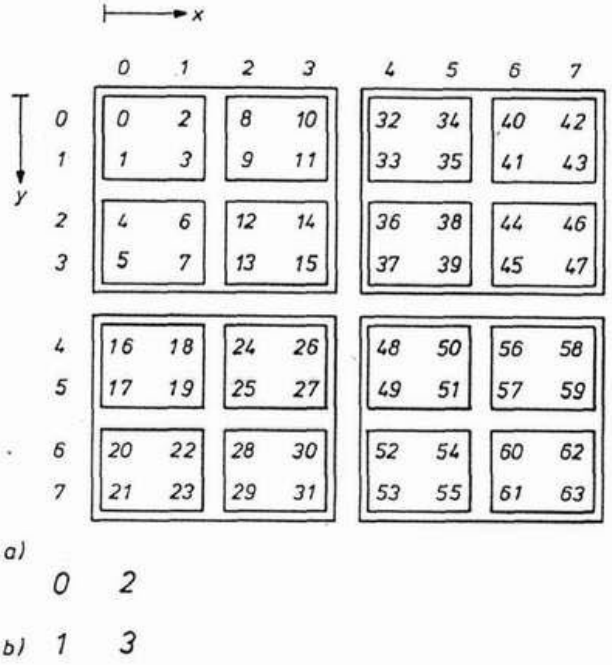


Fig. 6 Bitwise interlacing is a recursive coding scheme
a) codes for x, y = 0, 1, ..., 7
b) principial arrangement of codes and "blocks" of codes

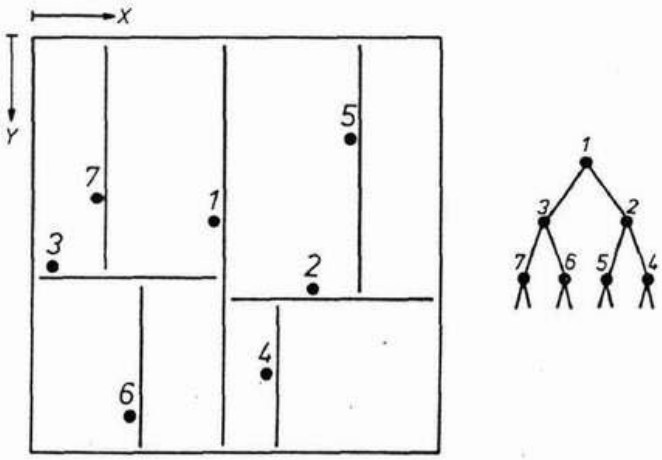


Fig. 7 The k-d tree structuring scheme

distribution: Data points with identical x values are automatically ordered according to y values, and vice versa. This is done without providing or handling any discriminators.

But the main advantage is, as already mentioned, that dynamic data structures can be built which allow to insert and delete multi-key records. In the next section we will briefly review the bacic data structures taken into consideration.

3 Structuring the Bitwise Interlaced Data

The bitwise interlaced keys, called "codes" or "record codes" in the following, are regarded as one dimensional data. So any data structure for storing single keys

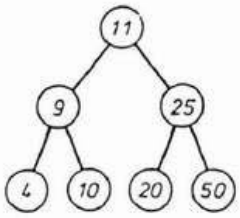


Fig. 8
A binary search tree

can be used. A sequential allocation of sorted codes is appropriate when no insertions and deletions of records must be performed. Binary search on this structure can be regarded as an implicit tree structure. To manage dynamically changing data volumes, we select the binary search tree (Fig. 8) making the tree structure explicit.

Although randomly built search trees have a good mean behaviour they may degenerate in the worst case yielding an O (N) insertion and deletion time. Therefore, we have actually used balanced trees. The classical AVL tree balancing mechanim is known best [5]. Actually we have chosen the recently developed 1-2 brother tree mechanism [6] which is conceptionally simpler. A 1-2 brother tree is a height balanced search tree, where each internal node has either two sons or, if it has one son, its brother has two sons. 1-2 brother trees have O (log N) worst case insertion and deletion time.

The range search algorithm described in the following chapter applies to both randomly built and balanced trees. The experimental results shown in chapter 5 are obtained by using 1-2 brother trees.

4 Range Search

In this section we describe the range query algorithm with bitwise interlaced keys (Fig. 5). To illustrate the algorithm, we discuss the two dimensional case (k = 2) in which all records have two keys. The search range is given by two records. By interlacing the keys of each record, one gets the minimum and the maximum code, thus defining the code range (see Fig. 9).

first approach of range query is now the same as in the two dimensional case. We suppose a binary search tree. Each node of this tree is associated with a record, represented by a record code. This record code is the result of interlacing the multidimensional keys of the record.

To show clearly the essence of the algorithms, special cases like "tree is empty" or "node is a leaf" are omitted in this paper.

Let P be the root node of the tree.

Range (P)

Case 1 [Recordcode (P) < Minimumcode]
Low subtree (containing only recordcodes less than recordcode (P)) is discarded. Perform RANGE (HISON).
(HISON is the son of P with recordcode greater than recordcode (P).)

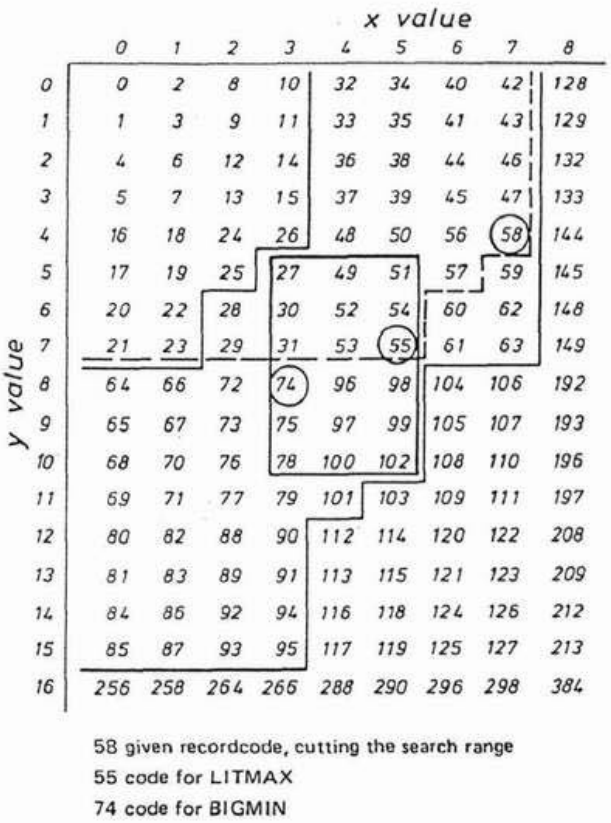


Fig. 9 Search range with range minimum code 27 and range maximum code 102

Case 2 [Recordcode (P) > Maximumcode]
High subtree (corresponding to low subtree) is discarded; Perform RANGE (LOSON).
(LOSON corresponding to HISON.)

Case 3 [Minimumcode <= Recordcode (P) <= Maximumcode]
If the record lies within the search range, report P; Perform RANGE (LOSON); Perform RANGE (HISON).

This is not a really effective algorithm because it does not consider the fact that many record codes can be between range minimum code and range maximum code without being within the search range. This is illustrated in Fig. 9 by two staircases. The codes above the upper staircase are less than the range minimum code, codes below the lower staircase are greater than the range maximum code. They are not within the search range. But all codes between the two staircases are potential candidates for range searching. In figure 9 the case is illustrated when a given record code (58) is between range minimum and range maximum code without being in the range.

We now can draw a staircase the upper codes of which are always less or equal the record code and the lower codes of which are always greater than the record code. This staircase, of course, is cutting the search range. If

we do so, we get two new codes within the range. One code being the greatest code (55) within the range which is less than the record code (named LITMAX) and the other being the lowest code (74) within the range but greater than the record code (named BIGMIN). Because in our binary tree all nodes in the left subtree have codes greater than the root node, one can update the code range for each subtree. Figure 9 shows that we only have to search between range minimum code and LITMAX for the low subtree and between BIGMIN and range maximum code for the high subtree, because it is safe to say that codes between LITMAX and BIGMIN do not lie within the range. In this way we can dynamically shrink the code range for each subtree, which is data dependent.

Now we can formulate a range search algorithm considering the dynamically code range shrinking. By dynamically code range shrinking, a subtree is visited by the algorithm if and only if one of its nodes might lie in the range. The computation of LITMAX and BIGMIN for a given code the staircase of which cuts the search range is described below.

Let P be the root node of the tree.

Start with RMIN = range minimum code and RMAX = range maximum code.

Range (P, RMIN, RMAX)

```
Case 1   [Recordcode (P) < RMIN]
         Perform RANGE (HISON, RMIN, RMAX).
Case 2   [Recordcode (P) > RMAX]
         Perform RANGE (LOSON, RMIN, RMAX).
Case 3   [RMIN < = Recordcode (P) < = RMAX]
IF       record lies within the search range
THEN     BEGIN report p;
                Perform RANGE (LOSON, RMIN,
                Recordcode (P);
                Perform RANGE (HISON, Record-
                code (P), RMAX)
         END
ELSE     BEGIN Compute LITMAX and BIGMIN
                Perform RANGE (LOSON, RMIN,
                LITMAX);
                Perform RANGE (HISON, BIGMIN,
                RMAX)
         END
```

In case 3, it is possible that a code range consists merely of one point and this point is found. In such a case, it is not necessary to visit any subtree. In our implemented algorithm we do not consider this case as it turned out to be very seldom and so we could spare to query this case.

Computing LITMAX and BIGMIN

First, the idea of computing LITMAX and BIGMIN is outlined. Second, the implementation is given in a decision table form.

The algorithm can best be understood as a binary search. At each step a bisection of the x-y plane is made. The bisection alternates in x and y direction. The situation is shown in Fig. 10 when searching LITMAX. There actually the x dimension is devided. All codes on the right of the dividing line are greater than any code on the left. Without loss of generality we assume that the x value of the record point is greater than the dividing x value xd.

Three cases are possible:

1. The range is entirely on the right (xmin > xd). Then the search continues on the right.
2. The range is entirely on the left (xmax < xd). Then the lower right corner, indicated by "X" in figure 10, is LITMAX.
3. The range is overlapping the dividing line (xmin < = xd < = xmax).
   Then LITMAX is either the "candidate point" (indicated by "*" in figure 10) or within the hatched region.
   (Here the "candidate point" is saved and the search continues on the right. If this search fails, the "candidate point" is LITMAX.)

The search for BIGMIN is along the same lines.

This binary search is implemented by bitwise scanning the codes of range minimum, range maximum and dividing record point (beginning at the most significant bit). The three bits are examined at each step according to the LITMAX and BIGMIN decision tables given below.

LOAD function in the tables means: set a bit pattern into the bits of a code associated with the actual dimension starting with actual bitposition (see Fig. 11).

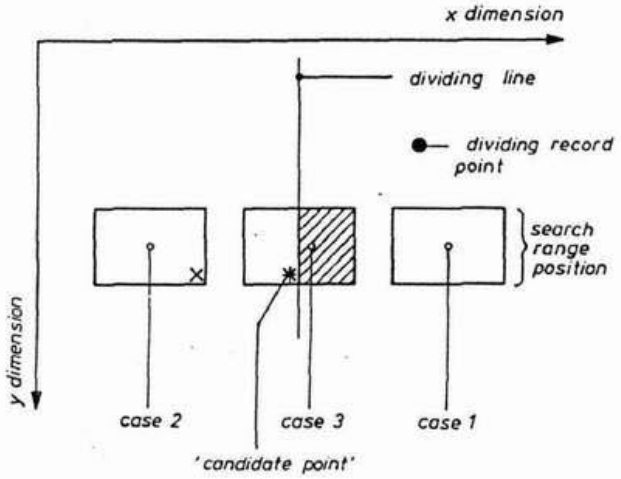


Fig. 10 Demonstration of the LITMAX computation

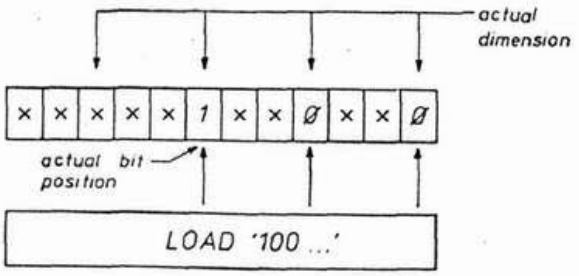


Fig. 11 LOAD Function

By modifying the LOAD function, the computation of LITMAX and BIGMIN applies also to any other interlacing scheme.

Let us now reconsider the effects of bitwise interlacing on range searching (for the two dimensional case): Comparing bitwise interlacing with the other extreme, the concatenation of keys (Fig. 1), one recognizes that in the latter case the dynamic code range shrinking leads to discarding narrow slices. Bitwise interlacing, however, leads to discarding areas, which is more adapted to searching for compact ranges.

5 Experimental Results

The experimental results presented in this section were obtained under the following conditions:

- 1-2 brother trees were chosen as data structure.
- The key values were produced using a pseudo random generator with uniform distribution.
- Both Fig. 12 and Fig. 13 show averages over 300 examples.
- Hypercube search ranges (which are the extreme of "compact ranges") are used in both figures.

In Fig. 12 for k = 2 the average number of records inspected, neglecting the number of records found, is depicted. From this figure we derive that the range searching is logarithmic with the number of records (N). In this experiment, the expected number of records to be found was held constant when increasing the total number of records (N). This was done by varying the range of key values, as shown.

Fig. 13 shows the surprising fact that the search effort decreases when increasing the number of dimensions.

Some experiments with Gaussian distribution led to approximately the same results.

6 Discussion

In [3] Bentley says: "perhaps the most outstanding open problem is that of maintaining dynamic k-d trees." In this paper we have presented a simple data structure with the following features:

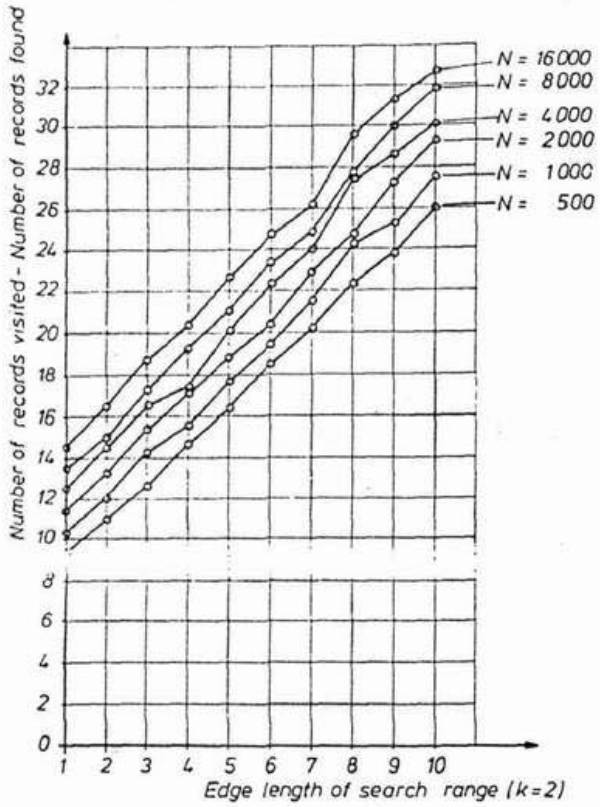
- Dynamically balancing; this results in logarithmic worst case insertion and deletion time. Exact matches are also performed in logarithmic worst case time.
- Any tree balancing mechanism can be used.
- Storage requirements are only two pointers per record to establish the search tree; no discriminators are needed.

LITMAX decision table

Actual bits of			
Dividing-record-code	Range-minimum-code (MIN)	Range-maximum-code (MAX)	Action
0	0	0	No action, continue
0	0	1	MAX = LOAD ("0111...", MAX); continue.
(0	1	0)	This case not possible because MIN < = MAX.
0	1	1	Finish.
1	0	0	LITMAX = Max; finish.
1	0	1	LITMAX = LOAD ("0111...", MAX); MIN = LOAD ("1000...", MIN); continue.
(1	1	0)	This case not possible because MIN < = MAX.
1	1	1	No action; continue

BIGMIN decision table

Actual bits of			
Dividing-record-code	Range-minimum-code (MIN)	Range-maximum-code (MAX)	Action
0	0	0	No action; continue.
0	0	1	BIGMIN = LOAD ("1000...", MIN); MAX = LOAD ("0111...", MAX); continue.
(0	1	0)	This case not possible because MIN < = MAX.
0	1	1	BIGMIN = MIN; finish.
1	0	0	Finish.
1	0	1	MIN = LOAD ("1000...", MIN); continue.
(1	1	0)	This case not possible because MIN < = MAX.
1	1	1	No action; continue.



N	Range of key values
500	0... 70
1000	0... 99
2000	0... 140
4000	0... 199
8000	0... 282
16000	0... 399

Fig. 12 Average number of records visited excluding the number of records found. N is the total number of records stored in the tree

- Logarithmic behavior is indicated by experiments: The results show that for small hypercube ranges the average number of records to be inspected is logarithmic with the number of records.

For both comparing the codes and computing LITMAX and BIGMIN, the computing time for each node is linear with k. Therefore, we assume that the expected time for searching small hypercube ranges is O (k (log N + F)), where k is the dimension, N is the number of records, and F is the number of records found.

A theoretical treatise of worst case and average complexity seems to be hard, especially when taking into account the "shape" of the search range. Perhaps some researcher will be encouraged by the experimental results to tackle this open problem.

The approach to range searching presented in this paper turns out to be simply realized with basic bit manipulations that call for assembly language programming. It is even easy to cast them into simple hardware realizations. This is especially interesting when constructing image analysis systems; in this field hardware support is a well introduced engineering practice.

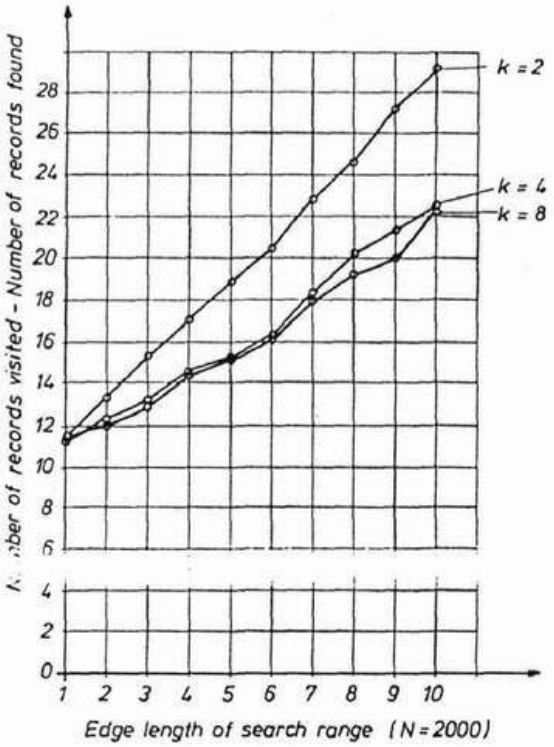


Fig. 13 Average number of records inspected for different dimensions. Range of key values for each k: 0 ... 140

Acknowledgement

The study reported on in this paper was carried out in the course of an image recognition project supported by the Bundesministerium für Forschung und Technologie, Fed. Rep. of Germany, under contract Nr. DV 5807-7.

References

[1] *H. Maurer, Th. Ottmann*: Manipulating Sets of Points, in M. Nagl. H.-J. Schneider (ed): Graphs, Data Structures, Algorithms. Carl Hanser Verlag, München Wien 1979, pp. 10 – 29

[2] *J. L. Bentley, H. H. Friedmann*: Data Structures For Range Searching. Computing Surveys, Vol. 11, No. 4, December 1979, pp. 397 – 409

[3] *J. L. Bentley*: Multidimensional Binary Search Trees in Database Applications. IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, July 1979, pp. 333 – 340

[4] *J. L. Bentley*: Multidimensional Divide-and-Conquer, Comm. of the ACM, April 1980, Vol. 23, No. 4, pp. 214 – 229

[5] *D. E. Knuth*: The Art of Computer Programming. Vol. 3: Sorting and Searching. Addison-Wesley Publishing Company, Reading, Mass., 1973

[6] *Th. Ottmann, D. Wood:* 1 – 2 Brother Trees or AVL Trees Revisited. The Computer Journal, Vol. 23 1980, No. 3, pp. 248 – 255.

Eingegangen im Dezember 1980