Experiments with a Closest Point Algorithm in Hamming Space

O. Nevalainen, J. Katajainen, University of Turku, Finnland

Key-words: *Search trees, multidimensional searching, best-match searching, data bases.*

Abstract: *A search tree structure originally introduced by Burkhard and Keller (Some Approaches to Best-Match File Searching, CACM Vol. 16, p. 230) is studied experimentally. The best-match searching time depends on the final minimal distance between the search point and the points in the search tree. Some modifications aiming at better performance are also proposed. For the purpose of comparison some tests with the well-known k-d tree are performed.*

Stichworte: *Suchbäume, mehrdimensionale Suche, „best-match"-Suche, Datenbanken.*

Zusammenfassung: *Eine Suchbaumstruktur, die ursprünglich von Burkhard und Keller (Some Approaches to Best-Match File Searching, CACM Bol. 16, p. 230) stammt, wird experimentell untersucht. Die „best-match"-Suchzeit hängt von der endgültigen Minimaldistanz zwischen dem Suchpunkt und den Punkten im Suchbaum ab. Es werden auch einige Abänderungen, die auf eine Leistungsverbesserung abzielen, vorgeschlagen. Zu Vergleichssprachen werden einige Tests mit dem bekannten k-d-Baum vorgenommen.*

1 Introduction

Let S be a set of n points in a k-dimensional Hamming space, i.e. the points are k-tuples of the form $x = (x_1, x_2, ..., x_k)$, where each component x_i belongs to a finite set C_i of say m_i different values. Then the distance of two points x and y in S is defined as the Hamming distance H(x, y) giving the number of components in which x and y differ:

$$H(x, y) = \sum_{i=1}^{k} d(x_i, y_i)$$

where

$$d(a, b) = \begin{cases} 1, \text{ for } a \neq b \\ 0, \text{ for } a = b. \end{cases}$$

Further, let $q = (q_1, q_2, ..., q_k)$ be a query point possibly not in S. The problem is to find a point in S nearest to q. This is the so-called *nearest neighbour, closest point* or *best matching point* of q.

The Hamming closest point problem is faced in some data base applications [2], [7]. Here the records correspond to the points of the Hamming space and the attributes of the records correspond to the components of the k-dimensional points. Given a query record one may wish to find a record in the data base closest to it in Hamming sense. This means that we are searching for a record with most matching attribute values. An other possible application of the closes point problem is met in the compression of formatted data files by spanning trees [3]. Here a static file is given as a spanning tree or a spanning forest where only the differences form the father nodes (records) are stored. In the construction of a minimal spanning tree we must repeatedly solve the closes point problem [1], [6] and it is evident that an efficient data structure supporting the searches is useful when writing an algorithm for the construction of a minimal spanning tree with the Hamming distance.

A large number of techniques solving the closest point problem or some of its many variants habe been developed and analyzed, for a survey see [5]. A common feature of these techniques is to try to take advantage of the geometric configuration of the point set. This in turn causes that their efficiency is in many cases dependent on the distance measure in use. Most analyses are for L_p-metrics, which are of the form

$$D_p(x, y) = [\Sigma |x_i - y_i|^p]^{1/p}.$$

In some cases, like in file searching, the distance of the objects is measured by Hamming distance which does not belong to the class of L_p-metrics.

In the present paper we make experiments with a particular data structure solving the closest point problem stated above. Burkhard and Keller [2] give three different data structures for the Hamming variation of the problem. A search tree (or forest) is constructed in such a way that each node classifies a subset of points

0013-5704/82/5 0277–05 $ 02.00/0

according to the distance to the root node. Branches which are too far away from the query point are excluded when searching for a closest point. If the preprocessing time, i.e. the time needed for constructing the search tree, is also included in the total processing time of the closest point application, the techniques (File structure 2 and 3 of [2]) using a forest are impractical. Even the experimental results of Burkhard and Keller favour the single tree approach (File structure 1) which is the object of interest in the present paper.

We start (Section 2) by introducing a Burkhard-Keller tree with a new parameter, mac-leaf-size, which serves as the stopping limit for the recursion in the construction of the subtrees. By using this parameter we can control the height of the tree to give a better performance. Theoretical analysis of the algorithm seems to be difficult and therefore only experimental results on the efficiency of the searching are given (Section 3). A comparison with the well-known k-d tree searching [4] is also made.

2 Burkhard-Keller Search Tree

The points of S are stored into the nodes of a k-ary tree T such that at the root level all nodes in the branch i are at the distance i form the root x^0, i.e. $H(x, x^0) = i$ for all x in the subtree T(x, i). The same action is repeated recursively until the size of a subtree is less than or equal to "max-leaf-size", which has a predefined fixed value. Such small subtrees are stored as linearly linked lists and the father node contains a pointer to the front of the list, see Fig. 1. A procedure constructing a Burkhard-Keller tree, called hereafter the B-K tree, is given in the appendix.

The running time of the algorithm depends on the ordering in which the points are given. In the worst case the construction of the tree demands an $0(n^2)$ time. One such case is when the tree degenerates to a linear list. It is easy to see that a B-K tree is listlike for a special point set S in which all distances are equal, i.e. H(x, y) is constant for all pairs of points x, y∈S. The time for a random set of points is, however, much

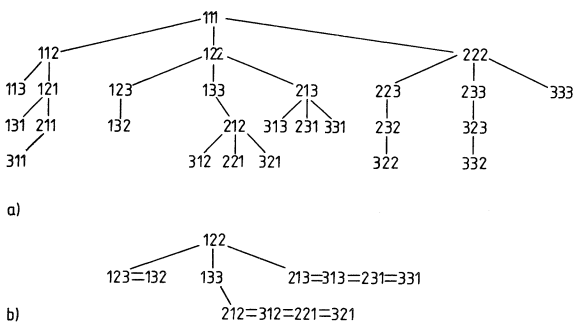

Fig. 1 a) A B-K tree of the point set S = {111, 112, 113, 121, ..., 333}; k = 3 "max-leaf-size" = 1.
b) Subtree "122" with "max-leaf-size" = 3.
——— branch links
═══ sequential links

smaller. This will be seen from the experiments with exact match searching (Section 3). Note that we need the same number of distance calculation while searching all the nodes in the tree and while constructing it.

The search for a best matching point of a query point q is made by first letting the initial quess of the minimal distance be $\xi = H(x^0, q)$. The search then proceeds in the branch ξ and the minimal distance is possibly improved in that branch. If a step downwards is no more possible a different branch at a greater distance from branch ξ is selected and the search continues there. That all nodes of the tree need possibly not be visited, follows from the so-called *joint cutoff criterion* [2] which is applied in the selection of the branches:

"If x^0 is the root of some subtree, then it is unnecessary to visit the branch t for which $|t\text{-}H(q, x^0)| \geqq \xi$."

The above condition may easily be derived from the triangle inequality. A procedure for searching the best matching point is given in the appendix.

3 Experiments

3.1 Burkhard-Keller tree

If one wants to apply the above file structure in a practical situation the following questions shoud be answered:

1) What is the expected percentage of the points in the search tree one has to visit while making a closest point search?
2) How should one select the value of the parameter "max-leaf-size" such that the searching time is minimal?

Table 1 The percentage R of the distance calculations as a function of m, the number of different values in a component
ξ, the minimal distance
k, the number of dimensionality (k = 10).

ξ \ m	m=2	m=4	m=6	m=20
0	0.9	0.5	0.5	0.7
1	2.9	7.5	7.9	7.7
2	7.1	18.8	25.1	51.1
3	13.3	44.0	59.6	90.6
4	21.8	69.1	85.3	98.4
5	33.3	84.1	95.7	100.0
6	49.3	93.1	99.1	100.0
7	65.2	97.7	100.0	100.0
8	81.1	99.5	100.0	100.0
9	99.7	99.9	100.0	100.0
10	100.0	100.0	100.0	100.0

A number of artificial point sets were generated with a pseudorandom generator. In each component the values were integers from the interval [1 : m]. Percentage R of the points visited is shown in Table 1*) for several

*) A DEC-10 computer with a KA10-processor was used in all tests. The programming language was FORTRAN-IV.

Table 2

The observed running time [sec] as a function of "max-leaf-size" for different m and ξ values; n = 1000; k = 10.

a) *B-K tree*

max-leaf-size		ξ 0	1	2	3	4	5	6	7	8	9	10
m=4	1	0.005	0.12	0.29	0.76	1.23	1.54	1.70	1.72	1.80	1.75	1.75
	9	0.005	0.06	0.15	0.34	0.52	0.64	0.69	0.72	0.73	0.73	0.72
m=6	1	0.006	0.13	0.42	1.02	1.54	1.64	1.80	1.80	1.77		
	5	0.005	0.07	0.25	0.58	0.82	0.92	0.95	0.94	0.95		
	9	0.006	0.06	0.19	0.43	0.58	0.68	0.70	0.72	0.71	0.71	
	20	0.009	0.06	0.21	0.42	0.56	0.61	0.62	0.63			
	50	0.015	0.04	0.25	0.51	0.60	0.64	0.73	0.65	0.64	0.62	0.58
	100	0.027	0.07	0.30	0.49	0.52	0.58	0.60	0.57	0.57	0.60	0.59
m=8	1	0.006	0.10	0.43	1.09	1.49	1.65	1.70	1.71	1.80	1.80	1.70
	9	0.006	0.04	0.22	0.46	0.61	0.67	0.69	0.70	0.69	0.70	0.70
m=10	1	0.007	0.13	0.54	1.28	1.67	1.73	1.77	1.73	1.69	1.71	1.80
	9	0.006	0.06	0.28	0.59	0.68	0.82	0.81	0.84	0.79	0.78	0.80

b) *k-d tree*

max-leaf-size		ξ 0	1	2	3	4	5	6	7	8	9
m=4	1	0.15	0.14	0.29	0.41	0.64	0.83	1.03	1.14	1.19	1.20
	16	0.05	0.05	0.13	0.22	0.33	0.36	0.38	0.39	0.40	0.40
m=6	1	0.09	0.12	0.18	0.42	0.59	0.79	1.02	1.14	1.19	1.21
	2	0.06	0.16	0.15	0.29	0.43	0.59	0.74	0.77	0.83	0.84
	4	0.05	0.06	0.10	0.23	0.33	0.42	0.51	0.53	0.55	0.55
	8	0.06	0.05	0.15	0.18	0.30	0.40	0.43	0.46	0.45	0.45
	16	0.02	0.05	0.14	0.20	0.31	0.37	0.40	0.40	0.40	0.40
	32	0.04	0.06	0.13	0.25	0.32	0.37	0.37	0.40	0.39	0.39
m=8	1	0.13	0.14	0.17	0.39	0.54	0.80	0.99	1.12	1.20	1.21
	16	0.02	0.06	0.12	0.20	0.30	0.35	0.37	0.38	0.39	0.38
m=10	1	0.08	0.09	0.19	0.36	0.54	0.76	0.98	1.13	1.19	1.20
	16	0.03	0.04	0.10	0.19	0.28	0.36	0.37	0.38	0.39	0.39

k, m, ξ combinations. The query points were generated from random points of the tree by changing the wanted number (ξ) of components in a point outside the range [1 : m]. These components were selected randomly, too. To find the minimal number of distance calculations we used in the tests the parameter value "max-leaf-size" = 1. Further the size of the point set was n = 1000 and for each (ξ, m)-combination the searching was repeated one hundred times. We observe that the number of distance calculations is very low for exact match queries (for $\xi = 0$, $R < 1$ %). When ξ increases the percentage strongly approaches 100 %.

In Table 2 we have the observed running time of the best match searching for some combinations of "max-leaf-size", m and ξ. The values in the table are means of 10,000 random search repetitions for $\xi = 0$ and means of 100 repetitions for $\xi \geq 1$. (Note that actually one should for $\xi = 0$ search once all the points in the search tree.) The modest number of repetitions for $\xi \geq 1$ causes a large variation in the observed running time. In addition the time measuring instrument of DEC-10 computer is rather inaccurate. From the results it is evident that for m = 6 the optimal "max-leaf-size" is ca. 5–9 for the case of exact match searches. For ξ-values greater than or equal to one a larger "max-leaf-size" gives still better results. A value of "max-leaf-size" chosen from the interval [5 : 20] is reasonable. This result looks natural if we remember from Table 1 that R reaches 100 % quite rapidly and then a linear list works best.

3.2 k-d tree

A further data structure for multidimensional searching is the k-d tree [4]. In it each internal node divides a hypercube of the coordinate space into two parts according to a discriminator value in a chosen coordinate direction. The discriminating value in each internal node is selected as the median value in the range of the chosen discriminator coordinate. Then all points with a coordinate value less than the discriminator value belong to the left subtree of the internal node and all points with a coordinate value greater than the value to the right subtree. The partitioning process is recursively repeated in the left and the right subtree until less than or equal to "max-leaf-size" points exist in the leaf hypercubes, Fig. 3.

When we are searching for the nearest neighbour of a query point we first recursively search for it in the subtree where the query point lies. When the searching in a subtree is finished we return to the root of the subtree and go to the other subtree if the hypercube

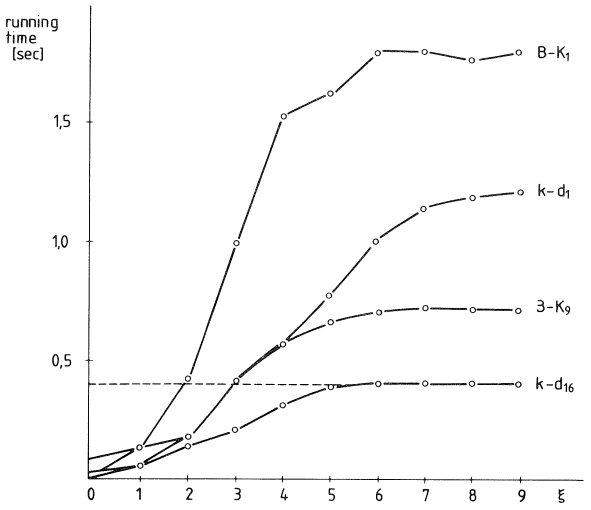

Fig. 2 The observed running time per one search operation for best match searches, a detail from Table 2; m=6, k=10, n=1000.

$B\text{-}K_1$: Burkhard-Keller tree, "max-leaf-size"=1
$B\text{-}K_9$: Burkhard-Keller tree, "max-leaf-size"=9
$k\text{-}d_1$: k-d tree, "max-leaf-size"=1
$k\text{-}d_{16}$: k-d tree, "max-reaf-size"=16
---: simple scanning

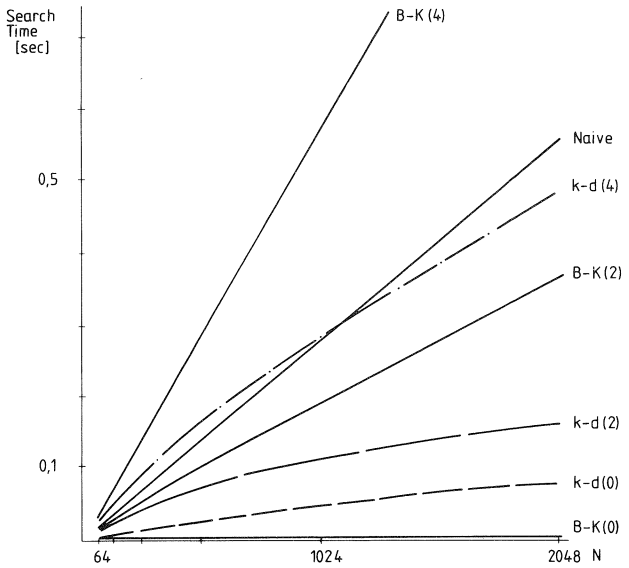

Fig. 4 Search time as a function of the number of points (n); m=6; k=10.
B-K (ξ): Burkhard-Keller tree, "max-leaf-size"=9;
k-d (ξ): k-d tree, "max-leaf-size"=16;
NAIVE: trivial method.

determined by it can contain points which possibly have more matching coordinates than the current nearest neighbour. As a matter of fact we must repeat the procedure for the whole k-d tree.

There are some possibilities to optimize a k-d tree [4]. First, in each hypercube the discriminator is selected from the coordinate in which the range of the values is largest. Second, "max-leaf-size" can be selected optimally. Because of implementation details we can use as the number of points and "max-leafsize" only values which are powers of two. In our tests the value 16 gave the best running time. Note that for a Euclidean case, 8 was the optimal value for "max-leaf-size" [6].

As a result of test runs Table 2b shows the running time as a function of "max-leaf-size" and the final distance ξ. The test situation was the same as for B-K trees.

3.3 Comparison of the results

When comparing the B-K tree and k-d tree (Tables 2a and b), we observe that a k-d tree is superior in the case studied. (For m = 6, ξ = 8, a B-K tree needs twice the time needed by k-d tree.) See also Fig. 4 for a graphical presentation of the results. The graphs of the running time look quite similar. It should be noted that the implementation of the k-d tree is more sophisticated: internal nodes of the k-d tree are stored as a heap whereas in the B-K tree a twoway linking is applied. Further the linking is avoided also in the leaves of the k-d tree;

Finally in Fig. 4 we show the observed average search time for a B-K tree and a k-d tree. For exact match queries the B-K tree works very well whereas for partial match queries the graphs indicate a rather weak running time. The same figure also shows the running time of a naive algorithm scanning all points. It is interesting to note that in this particular case for example for ξ = 4 the size of the point set must be at least 1000 that for a long run of closest point searches the k-d tree is profitable as to the running time. For a k-d tree the graph runs below that of naive solution, when $n \geqslant 64$ and ξ = 2.

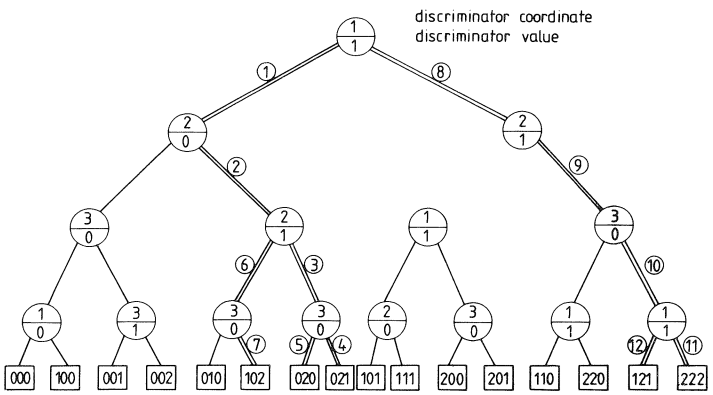

Fig. 3 A k-d tree for the point set {000, 001, 002, 100, 200, 222, 102, 201, 111, 110, 220, 020, 121, 021, 101, 010}. The steps when searching for the closest point of the point 122 are shown inside circles beside the lines. After step ④ ξ=2 and after ⑦ ξ=1. The nearest neighbour found is point 102.

4 Concluding Remarks

A new cutoff criterion in the construction of a multi-dimensional search tree by Burkhard and Keller [2] was introduced. This parameter aims at better performance by the well-known technique of using bucket-like leaf nodes in a tree. Experiments with random point sets indicate that the optimal leaf size depends on the final distance in the closest point queries. For increased final minimal distance a greater leaf size works better.

The advantage gained by the use of B-K trees is lost when the final minimal distance grows. This is due to the nature of Hamming distance which gives only little information from distance comparisons. For exact match queries the structure is very fast. By optimizing the storage structure of the tree it would surely be possible to decrease the time taken by a B-K tree. Further one can perhaps develop a reasonable technique to optimize the selection of root nodes when constructing the tree. An interesting open question is also the updating of the search tree.

Experiments with k-d trees show the power of the structure also in Hamming case. Like B-K trees, the method is sensitive to the final minimal distance but in comparison to the naive solution it gives an improved processing time also for rather large minimal distance.

APPENDIX PROGRAMS FOR A B-K TREE

```
      SUBROUTINE BUILD(KEY,LINK,ISON,N,K,LSIZE)
C     THIS SUBROUTINE CONSTRUCTS A BURKHARD-KELLER TREE OF A SET OF N
C  K-DIMENSIONAL POINTS (KEY(J,I),J=1,K I=1,N. THE MAXIMAL LEAF SIZE
C  IS GIVEN BY PARAMETER LSIZE. AT THE TERMINATION OF THE SUBROUTINE
C  THE ROOT OF THE TREE IS THE FIRST POINT (I=1) AND THE BRANCHES OF
C  THE TREE ARE GIVEN BY (ISON(J,I),J=0,K) FOR THE NODE I (I=1,N). IN
C  LEAF NODES LINK(I) CONNECTS THE POINTS OF THE LEAF, IN INTERNAL
C  NODES LINK(I)=0.
C     THE SUBROUTINE USES A VECTOR ISTACK(5) TO STORE THE ADDRESSES
C  OF THE SUBTREES FOR WHICH THE CONSTRUCTION IS NOT COMPLETED.
      DIMENSION KEY(10,1),LINK(1),ISON(0:10,1)
      DIMENSION ISTACK(500),NS(0:10),IQ(10)
C  THE POINTS INITIALLY FORM A LINEARLY LINKED LIST AND STACK
C  CONTAINS THE ADDRESS OF THE LIST.
      DO 2 I=1,N
      DO 3 J=0,K
3     ISON(J,I)=0
2     LINK(I)=I+1
      LINK(N)=0
      IND=1;ISTACK(IND)=1
C  THE MAIN LOOP, TERMINATE WHEN ALL SUBTREES HAVE BEEN PROCESSED.
4     IF(IND.EQ.0) RETURN
      IADR=ISTACK(IND); IND=IND-1; NEXT=LINK(IADR)
      DO 6 J=0,K
6     NS(J)=0
C  STORE THE POINT INTO THE SUBTREE GIVEN BY THE DISTANCE FROM
C  THE ROOT (IADR).
7     IF(NEXT.EQ.0) GO TO 13
      DO 12 I=1,K
12    IQ(I)=KEY(I,NEXT)
      J=ID(IQ,KEY,K,IADR); IA=LINK(NEXT)
      IF(NS(J).NE.0) GO TO 10
      ISON(J,IADR)=NEXT; LINK(NEXT)=0; GO TO 11
10    LINK(NEXT)=ISON(J,IADR); ISON(J,IADR)=NEXT
11    NS(J)=NS(J)+1
C  GET THE NEXT POINT.
      NEXT=IA
      GO TO 7
C  PUT INTO THE STACK THE BRANCHES TO BE PROCESSED.
13    LINK(IADR)=0
      DO 8 J=0,K
      IF(NS(J).LE.LSIZE) GO TO 8
      IND=IND+1; ISTACK(IND)=ISON(J,IADR)
8     CONTINUE
      GO TO 4
      END

      SUBROUTINE SEARCH(KEY,LINK,ISON,K,IQ,KSI,IMIN,ITOT)
C     THIS SUBROUTINE SEARCHES FOR A POINT (IQ(J),J=1,K) THE CLOSEST
C  POINT IN THE BURKHARD-KELLER TREE ((KEY(J,I),J=1,K),(LINK(I),
C  (ISON(J,I),J=1,K)),I=1,N). THE MINIMAL DISTANCE IS STORED IN
C  KSI AND THE CLOSEST POINT IS (KEY(J,IMIN),J=1,K). THE VARIABLE
C  ITOT GIVES THE TOTAL NUMBER OF DISTANCE CALCULATIONS.
C     THE PATH FROM THE ROOT TO THE CURRENT NODE OF THE TREE IS
C  STORED IN A STACK (IOPJ(I),IPNY(I),IPDEL(I),I=1,IND), GIVING THE
C  POINTER TO A NODE, THE DISTANCE OF THE QUERY POINT FROM THE NODE
C  AND THE CURRENT DELTA-VALUE.
      DIMENSION KEY(10,1),LINK(1),ISON(0:10,1)
      DIMENSION IQ(10),IPJ(1000),IPNY(1000),IPDEL(1000)
C  INITIALIZATIONS
      IROOT=1;ICURR=IROOT; KSI=K+1; IMIN=0; IND=0; ITOT=0
C  MUST WE FOLLOW THE LINK?
24    NEXT=LINK(ICURR)
      IF(NEXT.EQ.0) 36,35
C  STEP DOWN TO THE SON.
36    J=ID(IQ,KEY,K,ICURR); ITOT=ITOT+1
      IF(KSI.GT.J) 42,43
42    KSI=J; IMIN=ICURR
43    IDELTA=0; KP=J+IDELTA; GO TO 41
C  FOLLOW THE LINK
35    NEXT=ICURR
46    J=ID(IQ,KEY,K,NEXT); ITOT=ITOT+1
      IF(KSI.GT.J) 44,45
44    KSI=J; IMIN=NEXT
45    NEXT=LINK(NEXT)
      IF(NEXT.NE.0) 46,37
C  STEP TOWARDS THE ROOT.
37    IF(IND.EQ.0) RETURN
      IDELTA=IPDEL(IND); J=IPJ(IND); ICURR=IPNY(IND); IND=IND-1
40    IDELTA=-IDELTA
      IF(IDELTA.GE.0) IDELTA=IDELTA+1
      IF(ABS(IDELTA).GT.K) 37,38
C  ARE THERE ANY BRANCHES LEFT?
38    KB=J+IDELTA
      IF((KB.GT.K).OR.(KB.LT.0)) GO TO 40
C  DOWN THE BRANCH.
41    NEXT=ISON(KB,ICURR)
      IF(NEXT.EQ.0) 40,39
C  IS THE BRANCH TOO FAR AWAY?
39    IF(ABS(KB-J).GE.KSI) GO TO 37
C  THE SON IS REALLY WISITED, STORE THE CURRENT NODE.
      IND=IND+1; IPNY(IND)=ICURR; IPJ(IND)=J; IPDEL(IND)=IDELTA
      ICURR=NEXT; GO TO 24
      END

      FUNCTION ID(IQ,KEY,K,IADR)
C  THIS FUNCTION DETERMINES THE HAMMING DISTANCE OF THE POINT
C  (IQ(J),J=1,K) AND (KEY(J,IADR),J=1,K).
      DIMENSION IQ(10),KEY(10,1)
      ID=0
      DO 1 J=1,K
      IF(IQ(J).NE.KEY(J,IADR)) ID=ID+1
1     CONTINUE
      END

C  THE MAIN PROGRAM.
      DIMENSION KEY(10,1024),LINK(1024),ISON(0:10,1024),IQ(10)
C  THE MAXIMAL LEAF SIZE
      LSIZE=10
C  A RANDOM POINT SET. THE COMPONENTS ARE FROM THE RANGE 1 TO L
      N=100; K=10; L=6
      DO 1 I=1,N
      DO 2 J=1,K
2     KEY(J,I)=1+L*RAN(Z)
1     TYPE *,I,(KEY(J,I),J=1,K)
C  CONSTRUCTION OF A BURKHARD-KELLER TREE
      CALL BUILD(KEY,LINK,ISON,N,K,LSIZE)
100   CONTINUE
C  A QUERY POINT
      DO 3 J=1,K
3     IQ(J)=1+L*RAN(Z)
      TYPE *,(IQ(J),J=1,K)
C  SEARCHING
      CALL SEARCH(KEY,LINK,ISON,K,IQ,KSI,IMIN,ITOT)
      TYPE *,KSI,IMIN,ITOT,(KEY(J,IMIN),J=1,K)
      GO TO 100
      END
```

References

[1] *J. L. Bentley, J. H. Friedman:* Fast Algorithms for Constructing Minimal Spanning Trees in Coordinate Spaces. IEEE Trans. on Computer C-27. 2 (1978)

[2] *W. A. Burkhard, R. M. Keller:* Some Approaches to Best-Match File Searching. Comm. ACM 16, 4 (1973)

[3] *J. Ernvall, O. Nevalainen:* Compact Storage Schemes for Formatted Files by Spanning Trees. BIT 19, 4 (1979)

[4] *J. H. Friedman, J. L. Bentley, R. A. Finkel:* An Algorithm for Finding Best Matches in Logarithmic Expected Time. ACM Trans. Math. Software 3, 3 (1977)

[5] *H. Maurer, Th. Ottmann:* Manipulating sets of points – a survey. In: Graphen, Algorithmen, Datenstrukturen: Workshop 78. Hanser, München, Wien (1978)

[6] *O. Nevalainen, J. Ernvall, J. Katajainen:* Finding Minimal Spanning Trees in a Euclidean Space. BIT 21, 1 (1981)

[7] *G. Wiederhold:* "Database Design". McGraw-Hill Book Company, NY (1977)

Zweiteingang am 20.8.1981