Pattern Recognition, Vol. 23, No. 1/2, pp. 171–178. 1990
Printed in Great Britain.

0031-3203/90 $3.00 + .00
Pergamon Press plc
Pattern Recognition Society

STRATEGIES FOR EFFICIENT INCREMENTAL NEAREST NEIGHBOR SEARCH

ALAN J. BRODER*
The MITRE Corporation, 7525 Colshire Drive, McLean, VA 22102, U.S.A.

(*Received 29 September* 1988; *received for publication* 7 *February* 1989)

Abstract—Algorithms for determining the m nearest neighbors of a query point in k-dimensional space can be inappropriate when m cannot be determined in advance. We recast the m nearest neighbor problem into a problem of searching for the next nearest neighbor. Repeated invocations of an incremental searching algorithm allow an arbitrary number of nearest neighbors to be determined. It is shown that incremental search can be implemented as a sequence of invocations of a previously published non-incremental algorithm. A new incremental search algorithm is then presented which finds the next nearest neighbor more efficiently by eliminating redundant computations. Finally, the results of experimental computer runs comparing the two approaches are presented.

Nearest neighbor classification   Multi-dimensional searching   Search trees
Computational geometry   Algorithms

1. INTRODUCTION

The need to determine the m nearest neighbors of a query point from a collection of k-dimensional data points arises in many diverse applications. Examples of such applications include nonparametric classification and density estimation.[1] The use of "brute force" to determine the m nearest neighbors (computing and ordering the distances between the query point and all data points) can be prohibitive, especially when considering the cost of thousands of queries on data sets containing hundreds of thousands of data points.

If the data points are approximately uniformly distributed in k-space, a simple uniformly spaced k-dimensional grid structure can yield excellent results.[2] However, the measurement of natural or man-made phenomena will tend to produce non-uniform, highly clustered data. In such cases it is desirable to use a data structure which can adaptively partition the k-space, partitioning more finely in areas of higher data point density. A variety of data structures having this adaptive partitioning property have been proposed, and an excellent survey can be found in Ref. (3).

Bentley[4] introduced a multi-dimensional tree structure, the k-d tree, which allows for the efficient implementation of many types of associative queries. Friedman *et al.*[5] presented an algorithm that performs an m nearest neighbor search on a collection of N points stored in a k-d tree with expected time proportional to log N.

For many applications, it may not be known *a priori* how many of the nearest neighbors to a query point will be needed. Rather, it may be more desirable to incrementally perform a nearest neighbor search, with each invocation of the search routine returning the next nearest neighbor.

This paper describes two different strategies for implementing incremental nearest neighbor search algorithms on records stored in k-d trees. The first strategy involves repeatedly invoking a non-incremental algorithm. Because of the significant amount of redundant computation involved, this technique can be highly inefficient.

The second strategy presented in this paper involves a modified search algorithm along with auxiliary data structures for preserving the search state. The auxiliary data structure allows for an unlimited (subject only to memory constraints) number of searches from different query points being conducted in parallel.

2. k-d TREES

A k-d tree is a binary tree structure for storing and performing operations on records containing k-dimensional keys. Each key is a vector of k real or integer values. The key thus specifies the record's position in k-space.

The root of a k-d tree represents the infinite k-dimensional space containing the collection of records. Each node represents a sub-space containing a subset of the collection of records. The two children of a non-terminal node are obtained by partitioning the node's sub-space into two parts by a hyper-plane perpendicular to one of the k coordinate axes (the discriminating axis). The position of the hyper-plane

* Supported by MITRE Sponsored Research Project: Spatial Data Base Technology.

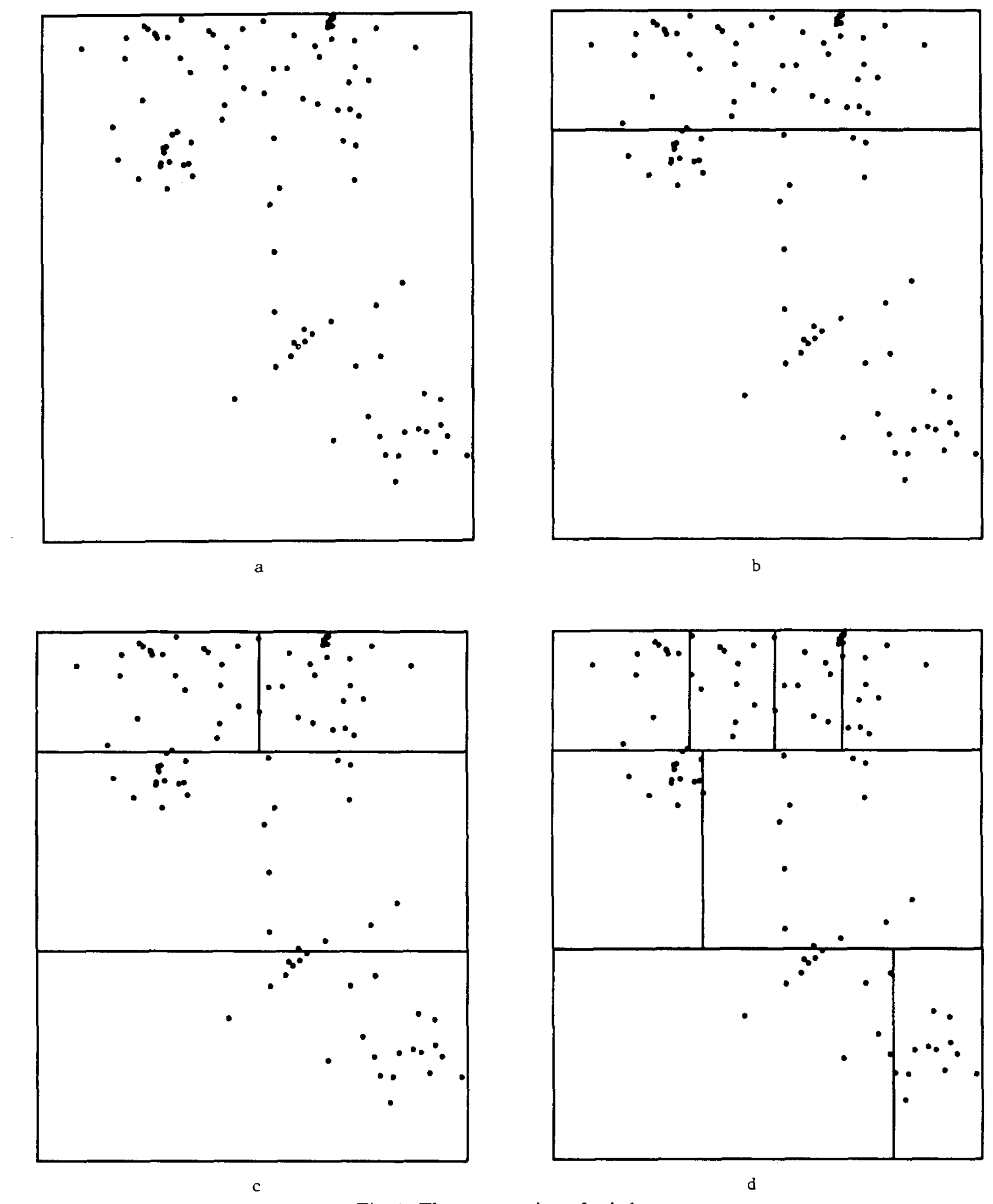
Fig. 1. The construction of a k-d tree.

is chosen so that each of the children contains approximately half of the parent's records. The discriminating axis is determined by choosing the axis along which the parent's records are most dispersed (see Ref. (5) for a discussion of methods for measuring the dispersion). Children that contain less than a threshold number of records (the bucket size) become the leaf nodes in the tree.

Figures 1(a)–1(d) illustrate the steps in building a k-d tree on a small 2-dimensional data set. The root of the tree is represented by the box drawn surrounding the record points in Fig. 1(a). The left and right children of the root are the smaller boxes each surrounding half the records, as shown in Fig. 1(b). The remaining figures show successively deeper levels of the tree represented in a similar fashion.

3. INCREMENTAL SEARCH

3.1. *Nearest neighbor classification*

Friedman *et al.*'s algorithm *SEARCH*[5] finds the m nearest neighbors of a point from a collection of N points stored in a k-d tree. The requirement that m be specified in advance of the search can be very restrictive in applications such as non-parametric classification.

To illustrate this, consider a collection of records whose keys are k element vectors in a k-dimensional feature space. Further, assume that each of the records has been assigned a label $\mathscr{L}_i$, from among l possible labels. Given a new k element record, R_k, it is desired to assign a label to R_k based in some way on the labels already assigned to its nearest neighbors in k-space. The m nearest neighbor rule[1] assigns to R_k the most frequently represented label from among R_k's m nearest neighbors. Using *SEARCH*, all of R_k's m nearest neighbors will need to be retrieved to determine the "winning" label. However, if the m nearest neighbors could be retrieved one at a time (ordered by distance from R_k), then it might be possible to determine the "winning" label well before m neighbors have been retrieved. Likewise, it is possible that after the m nearest neighbors have been retrieved using *SEARCH*, no single label has a significant plurality. In this case, it may be desirable to continue retrieving additional nearest neighbors until a higher level of confidence is attained. *SEARCH* is clearly inappropriate for this problem, since all $m + n$ nearest neighbors must be retrieved in order to obtain the n additional nearest neighbors beyond the already retrieved m nearest neighbors.

Ideally, we would like to treat the collection of N records as if it were formed into a list, ordered by distance from the query key. The nearest neighbor search process would then consist of removing records one at a time from the head of the list. However, rather than construct and order the entire list, a function can be defined which in conjunction with a small amount of auxiliary storage effectively encodes the fully ordered list. Thus, each time the function is called, the next record in the implicit list is returned; the auxiliary storage contains just enough information so that the following record in the implicit list can be identified on the next call. A function operating in this way is known as a *generator* and the auxiliary storage is known as the *seed*.[6,7] An initialization function is used in conjunction with the generator, the sole purpose of which is to initialize the seed data structure before the generator is called the first time.

3.2. *A naive algorithm*

A generator *naive-search* which incrementally returns the next nearest neighbor upon each invocation can be implemented using repeated calls to the non-incremental algorithm *SEARCH*. To do this, simply use *SEARCH* to compute an ordered list of the first c nearest neighbors. Each call to naive-search then returns the next record in the list. Each time the list is exhausted, *naive-search* calls *SEARCH* again, multiplying its parameter m (the number of neighbors to find) by the same constant c.

Naive-search maintains no state information about previous searches, and thus can be extremely inefficient due to the many nearest-neighbor computations that are repeatedly performed. To illustrate this, and without loss of generality, consider the case where *naive-search* has been called n times to return the first n nearest neighbors, where n is a power of c (i.e. $n = c^i$). Then the total number of nearest neighbors which will have been retrieved by *naive-search* through multiple calls to *SEARCH* is

$$n + \frac{n}{c} + \frac{n}{c^2} + \frac{n}{c^3} + \cdots + c. \qquad (1)$$

Note that when the user calls *naive-search* just one more time to request the $(n + 1)$th nearest neighbor, *SEARCH* must be called again to find the first c^{i+1} neighbors (in addition to all those previously computed)! Figure 2 illustrates the behavior of *naive-search* by plotting (for $c = 2$) the total number of records retrieved to incrementally find the nth nearest neighbor.

The next section describes a much more efficient algorithm which avoids all redundant nearest neighbor calculations by preserving the search state between invocations.

3.3. *An efficient incremental search algorithm*

We will now describe a time and space efficient generator algorithm which performs incremental nearest neighbor search. To that end we will need to describe both the initialization function *search-init* and the generator function *search-next*. As the development proceeds we will also describe the contents of the seed data structure.

The function of *search-init* is to initialize the seed for a nearest neighbor search. *Search-init* is passed a pointer to the *k*-d tree containing the collection of

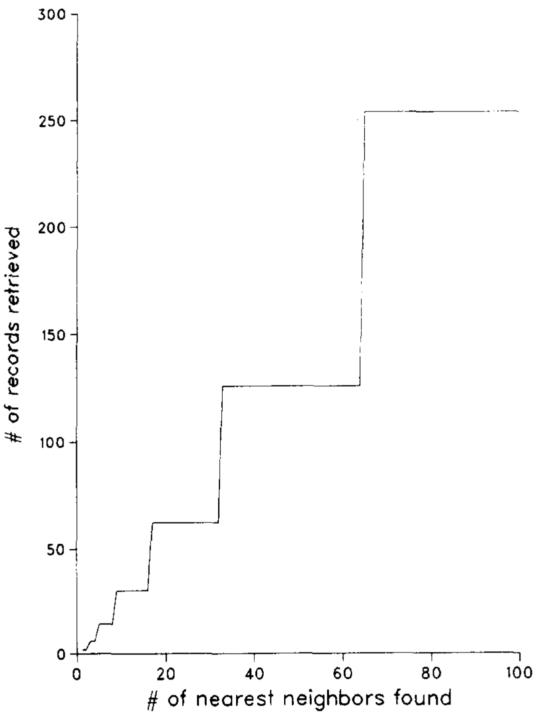

Fig. 2. Total number of records retrieved by *naive-search* to incrementally find the *n*th nearest neighbor.

records, and a pointer to a query key specifying a location in k space. The pointers to the tree and to the query key are stored in the seed. *Search-init* recursively descends the tree and locates the leaf node which spatially contains the query key. A list of pointers to the non-terminal nodes that were traversed during the descent is then stored in the seed (we call this the recursion trace).

Search-next can now use the information in the seed to locate the first nearest neighbor to the search-key. The Euclidean distance is computed between the query key and each of the records contained in the leaf node pointed to by the head of the recursion trace. The pointers to the records become elements of a queue which is ordered by distance to the query key.

The record at the head of the queue is now a possible candidate for the nearest neighbor. In fact, the distance r from the query key to this record will serve as an upper bound for the distance to the actual nearest neighbor. Any subsequent candidate for nearest neighbor must lie within the hyper-sphere S_r centered at the query key with radius r. If S_r is completely enclosed by the hyper-volume of the leaf node, then it is certain that the first record in the candidate queue must be the nearest neighbor (since all the records within the leaf have already been examined). If S_r is not contained entirely within the leaf node, then it is possible that a record closer than r may lie on the other side of one of the leaf node's bounding hyper-planes.

To ensure finding the correct result the tree is ascended one level by removing an element from the head of the recursion trace stored in the seed. New candidate records are determined by recursively descending the other child of the new non-terminal node. However, it is only necessary to consider records contained in leaf nodes that spatially intersect S_r (this is known as the *bounds-overlap-ball* test[5]). The distances between the new candidate records and the query key are computed and the records are then merged into the candidate queue. The first record of the queue is now the new best candidate for nearest neighbor and defines a new value of r.

This process of ascending the tree and evaluating new candidate records is repeated until S_r (whose radius is monotonically decreasing as the process continues) is entirely enclosed by the space defined

```
(defun search-next (seed &aux key queue trace)
   (setq key   (get-seed-value seed KEY))
   (setq queue (get-seed-value seed QUEUE))
   (setq trace (get-seed-value seed TRACE))
   (descend-tree key (current-node trace) (bounds-of-current-node trace))
   ; while there is more left on the recursion trace and either the queue
   ; is empty or the ball is not within bounds
   (while (and (cdr trace)
               (or
                 (not queue)
                 (not (ball-within-bounds key (current-node trace)
                                              (distance-to (car queue))))))
      (setq trace (cdr trace)) ; pop the recursion trace
      (descend-tree key (other-child trace) (bounds-of-other-child trace)))
   (set-seed-value seed QUEUE (cdr queue))
   (set-seed-value seed TRACE trace)
   (car queue))

; Returns t if all the records beneath this node have been evaluated. As the
; recursion returns, this enables internal nodes on the examined-list to be
; combined.
(defun descend-tree (key node bounds &aux leftp rightp)
   (cond
      ((on-examined-list node) t)
      ((not (bounds-overlap-ball key bounds (distance-to (car queue)))) nil)
      ((leafp node)  (setq queue
                            (merge queue (build-queue node key) 'order-func))
                     (add-to-examined-list node) t)
      (t (setq leftp (descend-tree key (LEFT node) (bounds-of (LEFT node))))
         (setq rightp (descend-tree key (RIGHT node) (bounds-of (RIGHT node))))
         (cond ((and leftp rightp)
                  (remove-from-examined-list (LEFT node) (RIGHT node))
                  (add-to-examined-list node) t)))))
```

Fig. 3. *Search-next* in FranzLisp.

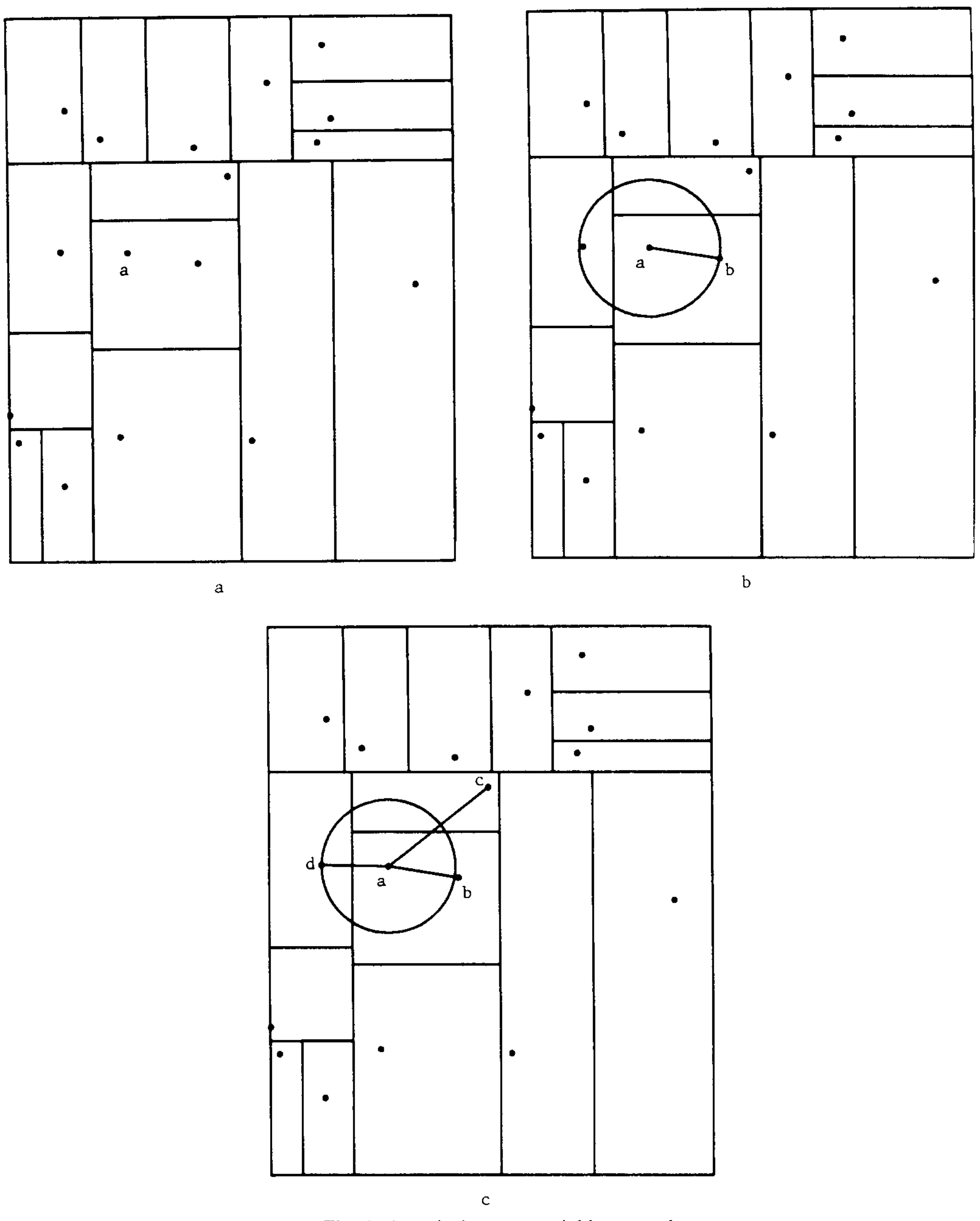

Fig. 4. A typical nearest neighbor search.

by the current non-terminal node. At this point, it is guaranteed that all the relevant records have been examined, and *search-next* can return the record at the head of the candidate queue. Before returning, *search-next* saves the remainder of the candidate queue in the seed. The value of r is also saved in the seed.

On the next call to *search-next*, the search for the next nearest neighbor can proceed in a similar fashion. The first element in the candidate queue is now the current best candidate for the next nearest neighbor and defines a new hyper-sphere $S_{r'}$ of radius r' (which by definition is at least as large as S_r). The current sub-tree is recursively descended to locate new candidate records contained in leaves which intersect $S_{r'}$ but not S_r. To make this process efficient, a list is maintained in the seed (the examined list) of all sub-trees whose records have been completely exhausted. The examined list thus enables the recursive descent to avoid redescending sub-trees that cannot contribute new records to the candidate list.

Once again, as before, the process of ascending the tree and obtaining new candidate records is then repeated until $S_{r'}$ is entirely contained within the space defined by the current non-terminal node.

Figure 3 presents a fragment of FranzLisp code that implements the algorithm described in this section. A complete listing of the FranzLisp code for all the algorithms mentioned in this paper is available by writing to the author.

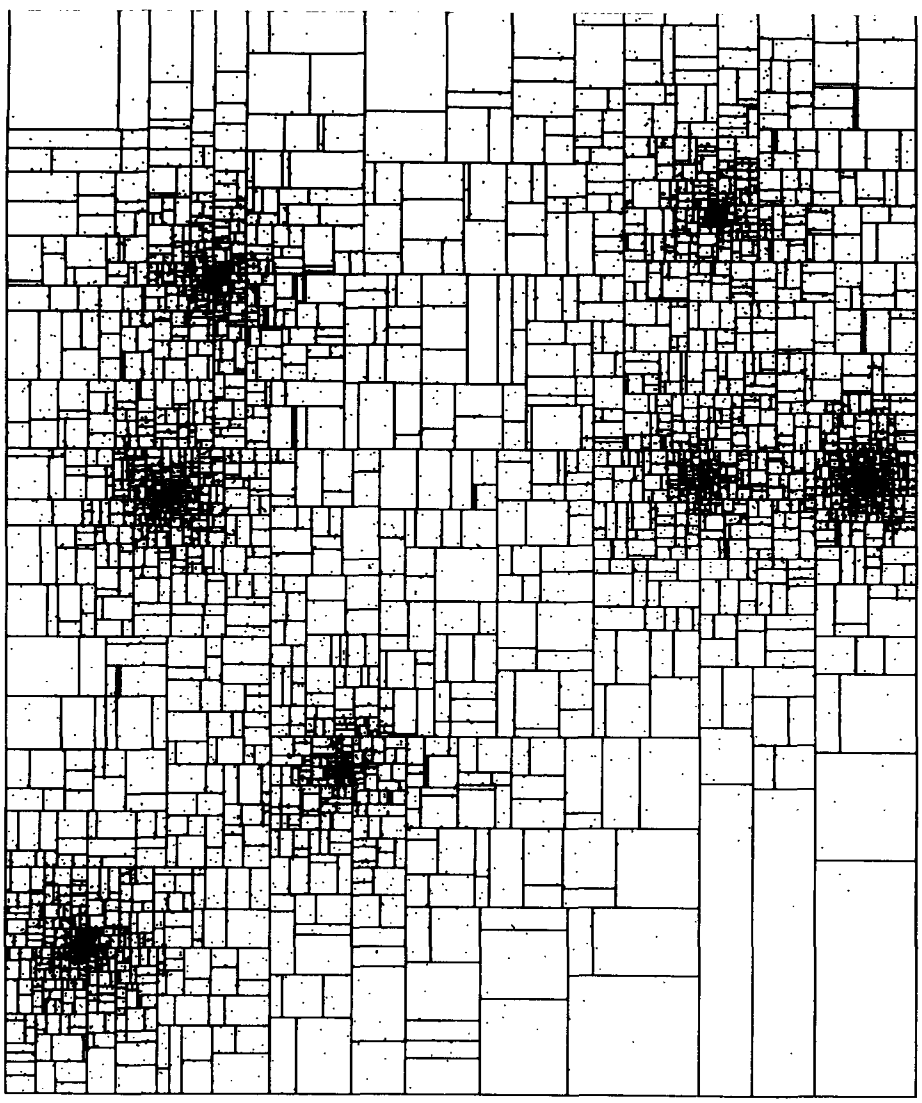

Fig. 5. The test data set.

3.4. *An example*

Figures 4(a)–4(c) illustrate the search algorithm on a simple two-dimensional example. Figure 4(a) shows a k-d tree with a bucket size of 1 for a collection of 16 records. The query point location is labeled a. The algorithm begins by descending the tree and locating the leaf node which spatially contains the query point. Figure 4(b) shows that the distance has now been computed from the query point a to the record labeled b in the same leaf. Record b is now our best current candidate for the nearest neighbor to a. The tree is ascended and other branches of the tree are explored in order to locate the two other leaves that overlap the circle defined by the line from a to b (denoted by $\overline{ab}$). In Fig. 4(c), the distances from a to the records c and d contained in those leaves are then computed. Since the length of $\overline{ad}$ is smaller than the length of $\overline{ab}$, record d is our new candidate for nearest neighbor to a. In fact, we can return record d as the actual nearest neighbor since the circle defined by $\overline{ad}$ does not overlap any as yet unexamined leaves. Records b and c are saved on the candidate queue.

On the next call to *search-next* record b would be returned since the circle defined by $\overline{ab}$ also does not overlap any as yet unexamined leaves. However, on the third call to *search-next*, the circle defined by $\overline{ac}$ does overlap several unexamined leaves. As before, the distances to the records contained in those leaves will have to be computed before the next nearest neighbor can be determined.

3.5. *Performance improvement*

For each call to *search-next*, the additional leaf nodes that are ovelapped by $S_{r'}$ have to be located. Some of these leaf nodes will be children of sub-trees that have already been partially explored. As a result some non-terminal nodes will have the ***bounds-overlap-ball*** test applied to them many times (possibly as many times as there are leaves beneath the node).

However, for a particular call to *search-next* no non-terminal node is tested more than once. This suggests that the performance of the algorithm will be improved if the number of repeated visits to non-terminal nodes can be reduced.

This can be achieved by modifying *search-next* to "look ahead" for the next *i* nearest neighbors rather than just the next 1 nearest neighbor. The first time *search-next* is called, a candidate queue is built using a hypersphere whose radius is determined by the distance from the key to the *i*th entry in the queue. The tree is ascended, as described above, until the hypersphere is entirely enclosed by the current internal node. We are then assured that the first *i* entries in the candidate queue are the *i* nearest neighbors. *Search-next* then returns the first element in the queue.

An entry in the seed allows *search-next* to keep track of how many valid nearest neighbors remain at the head of the queue. On subsequent calls to *search-next*, if the valid entries in the queue have been exhausted, then the above process is repeated to find the next *i* nearest neighbors. In any case, *search-next* returns the first entry in the queue after decrementing the valid nearest neighbor counter. The effect of this simple modification is that the number of *bounds-overlap-ball* tests on non-terminal nodes is reduced by as much as a factor of $1/i$. Another result of this "look-ahead" technique is that the computation time is greatly dominated by the non-redundant record-to-record distance computations.

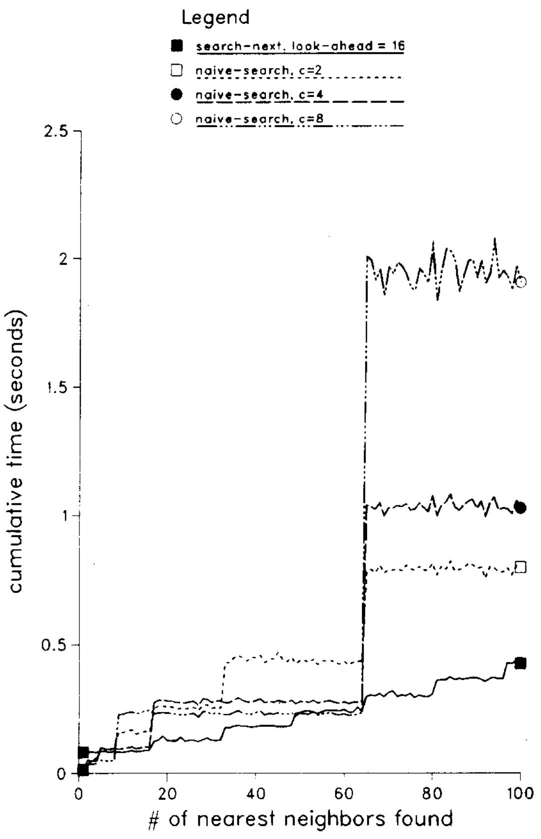

Fig. 6. Experimental data for first 100 neighbors.

4. EMPIRICAL ANALYSIS

Computer experiments were designed to measure and compare the performances of *naive-search* (for varying constant values of *c*) and *search-next*. For this experiment, the algorithm was coded in the C programming language and run on a Sun-3/180 under the SUN OS3.4. To obtain consistent results, both *naive-search* and *search-next* employ the identical underlying C code for *k*-d tree manipulation, access, interrogation, and distance computations.

The synthetic test data set for these experiments consisted of 10,000 2-dimensional records. Each record was represented as a pair of double precision floating point values. The records were randomly drawn from a non-uniformly distributed population of 2-dimensional points. The distribution function was synthesized by the superposition of seven bivariate normal distributions. Figure 5 shows a graphical display of the 10,000 record data set, overlaid by the computed *k*-d tree.

The CPU time used by the two algorithms was measured for numbers of nearest neighbors, *m*, ranging between 1 and 100, and for several different values of the *naive-search*'s constant *c*. For each value of *m*, a group of 100 query keys was drawn from a uniform random distribution covering the space enclosing the test data set.

The total amount of CPU time respectively used by *naive-search* and *search-next* to find the first through the *m*th nearest neighbors was measured for each of the query keys. The CPU times for each group of queries were averaged to produce estimated expected CPU times for each value of *m*. Figure 6 plots the results of these experiments, and dramatically demonstrates the increased performance of *search-next*.

5. CONCLUSION

Two algorithms have been presented which perform incremental nearest neighbor search on records stored in a *k*-d tree. It is experimentally shown that an algorithm employing a simple state-preservation data structure can significantly outperform an algorithm constructed of calls to a non-incremental algorithm.

REFERENCES

1. R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, Chapter 4. Wiley-Interscience, New York (1973).
2. L. J. Kitchen and M. Callahan, Optimal cell size for efficient retrieval of sparse data by approximate 2d position using a coarse spatial array, *Proc. IEEE Comput. Soc. Conf. Comput. Vision Pattern Recognition* 357–361 (1986).
3. H. Samet, The quadtree and related hierarchical data structures, *ACM Comput. Surv.* **16**, 187–260 (1984).
4. J. L. Bentley, Multidimensional binary search trees used

for associative searching, *Commun ACM* **18**, 509–517 (1975).

5. 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 (1977).
6. E. Charniak, C. K. Riesbeck and D. W. McDermott, *Artificial Intelligence Programming*, pp. 136–138. Lawrence Erlbaum, Hillsdale, NJ (1980).
7. W. Teitelman, *Interlisp Reference Manual*. Xerox Palo Alto Research Center (1978).

About the Author—ALAN JAY BRODER received the B.S. degree, Cum Laude, with honors in Computer Science from the University of Maryland, College Park in 1980. He received the M.S. degree in Computer Science from Columbia University in 1982.

Mr Broder has been on the staff of Advanced Technology Systems, where he conducted research in the areas of computer-assisted photogrammetry and advanced visual simulation systems. Subsequently, he was on the research staff at CGR Medical Corporation where he investigated and developed techniques for the compression, display, and interactive analysis of medical imagery.

Since 1984, Mr Broder has been with the MITRE Corporation, McLean VA, where he is currently Lead Scientist in the Image Processing Technology Center. His research at MITRE has been primarily in the areas of Hypermedia presentation systems, advanced workstation architectures, stochastic image synthesis, and spatial data base technologies.