SLAC-186
STAN-CS-75-513
UC-32

A SURVEY OF TECHNIQUES FOR FIXED RADIUS NEAR NEIGHBOR SEARCHING

JON LOUIS BENTLEY

STANFORD LINEAR ACCELERATOR CENTER

STANFORD UNIVERSITY

Stanford, California 94305

PREPARED FOR THE
U. S. ENERGY RESEARCH AND DEVELOPMENT ADMINISTRATION
UNDER CONTRACT NO. E(04-3)-515*

August 1975

*This work also supported in part by National Institutes of Health under Grant 1-P07-RR00898-01.

ABSTRACT

This paper is a survey of techniques used for searching in a multidimensional space. Though we consider specifically the problem of searching for fixed radius near neighbors (that is, all points within a fixed distance of a given point), the structures presented here are applicable to many different search problems in multidimensional spaces. The orientation of this paper is practical; no theoretical results are presented. Many areas open for further research are mentioned.

KEY WORDS AND KEY PHRASES

associative searching

k-d trees

fixed radius near neighbor searching

nearest neighbor searching

TABLE OF CONTENTS

1. Introduction

This paper deals with searching in a multidimensional metric space. We consider specifically the problem of searching for fixed radius near neighbors, that is, all points within a fixed distance r of a given point. Though our primary examination of the problem will take place in Euclidean k space, the constructs we examine will be applicable to an arbitrary multidimensional metric space. There are many ways of viewing the fixed radius near neighbor problem. The "online" problem first allows the procedure to store the points in a data structure then repetitively asks for all near neighbors within a given radius r of given "query points". The batched query approach assumes that the queries both arrive and are to be answered in groups, or "batches". The "all pairs" problem, given a collection of n points, asks for an enumeration of all pairs of the n within distance r of one another. We will examine techniques for dealing with all of these questions.

There are many applications of fixed radius near neighbor searching. These arise, in general, when an agent has the potential of affecting the state of all objects within a certain distance. Levinthal (11) used a fixed radius nearest neighbor search in his interactive computer graphics study of protein molecules. Since the interactions of the atoms within the molecule drop off so quickly with increasing distance, he approximated the forces acting on any given atom by considering only the forces caused by atoms within five angstroms of each other. An air traffic control system might be interested in locating all planes within, say, ten miles of each other (here a metric would be used in which one thousand vertical feet would be equivalent to many horizontal miles.) Iverson gave the fixed radius near neighbor problem as an excercise in his book A Programming Language (8,ex. 4.8(c)).

This paper is an attempt to provide a survey of the methods used to deal with the fixed radius near neighbor problem. As such, it contains no new results. The author has tried to gather together many different problems and techniques from different areas for a systematic presentation of the state of the problem.

Throughout this paper we will assume that we are dealing with a collection of n points in k dimensional Euclidean space. The fixed radius within which we are searching for near neighbors will be denoted by r. Except where otherwise noted, the distance function between two points will be the Euclidean distance. All of the structures developed will be extendable to other metrics such as the "city block" and the maximum coordinate metrics. When referring to a point x in k space, x's value in the i-th dimension will be written xi.

The fixed radius near neighbor problem is very similar to the nearest neighbor problem. The nearest neighbor in a set to a given point is defined to be that point in the set which is closest to the given point. Many of the notions discussed in this work were developed in connection with nearest neighbor algorithms. A great deal of work has appeared recently on the nearest neighbor problem. The interested reader is referred to Friedman, Baskett and Shustek (4), Friedman, Bentley and Finkel (5), Yuval (14), and Fukunaga and Narendra (6). The techniques described in this paper will be applicable to the nearest neighbor problem as well as many other search problems in multidimensional spaces.

Section 2 of this paper examines the online problem. It is here that the data structures used throughout the paper are defined. The batched query model is considered in section 3. In section 4 the all pairs problem is investigated. Section 5 considers what actions are appropriate when the points in the space are objects in motion. Some of the details of implementation of the algorithms are covered in section 6. Section 7 discusses the many areas open for further work in this problem.

2. The Online Model

As the online model applies to the fixed radius near neighbor problem, the search procedure is initially given the collection of n points and sufficient time to organize the points into a suitable data structure. After that the system is repetitively asked to search for all fixed radius near neighbors to given "target points" which may or may not be among the original collection of n points. The radius r may vary among queries. It is important to specify approximately how many queries will be posed during the lifetime of the system. If only some very small number of queries will be made, then it will probably not pay to impose any expensive structure on the data (unless fast response is critical). On the other hand, if an extremely large number of queries will be made, then it would be cost effective to perform some very elaborate and expensive structuring of the data. In this discussion we will assume that the number of queries will be approximately n.

We will now proceed to examine various data structuring schemes suitable for the fixed radius near neighbor problem. We will investigate both the structures themselves and the corresponding search algorithms. Figures 1 thru 4 will be used to illustrate the brute force, projection, cell, and k-d tree techniques, respectively. In figures 2 thru 4, the solid lines represent the partitioning of the space. In all the figures, the diagonally striped regions are those examined when searching for the target point, which is marked with an x. The meaning of the figures will be obvious, given the description of the structures which follows in sections 2.1 thru 2.4.

2.1 Brute force

The simplest approach to the fixed radius near neighbor problem is to store each of the n points in an array, list, or some other simple sequence. As each query arrives, all members of the list are scanned and all fixed radius near neighbors are enumerated. This technique involves linear storage for the structure, preprocessing time linear in n, and each query is answered with n distance calculations. Though this technique is unsophisticated in that it performs many distance calculations, the overhead is quite small. For small point sets (especially in high dimensionality spaces), the brute force approach will be hard to beat.

2.2 Projection

The projection technique is referred to as inverted lists by Knuth (9). This technique was applied by Friedman, Baskett and Shustek in their solution of the nearest neighbor problem (4). Projection involves keeping, for each dimension, a sequence of the points in the space sorted by that dimension. (A plex of pointers to the elements of the point set is probably the most favorable implementation--we will see below that binary searches should be easily accomplished.) These k lists can be obtained using some standard sorting algorithm in time of O(k*n*log n), and stored using O(n*k) words of storage. After the preprocessing, a query for all fixed radius near neighbors to point x can be answered by the following search procedure: Choose a dimension, say the i-th. Look up x's position in the i-th sequence, using a binary search (of cost O(log n)). Now scan down the list in decreasing order until a record is found whose i-th key is less than xi-r, and scan up the list until a record is found whose i-th key is greater than xi+r (note that these scans could be made using a binary search). All points in the collection within distance r of point x must be in the list between the two extremal points just found.

More should be said about how the i-th dimension is chosen. If the points are fairly randomly distributed among all dimensions, then one sorted sequence is all that really needs to be kept. If, however, there is much clustering in certain parts of the space, then it can pay to keep sorted sequences of all k dimensions. To choose which one to use for a given query, the local density around the point x in each dimension could be examined, and the sparsest dimension would be chosen. One implementation of this philosophy would be to calulate all the extremal pairs by doing binary searches for xi-r and xi+r (in O(k*log n) time), then choose the dimension which has the fewest points between the extremal pair. It might be worthwhile to spend some preprocessing time to determine how many of the dimensions should be kept as sorted sequences; if the points are bunched closely together in a particular dimension, then it would rarely be chosen.

The projection method is quite efficient for point sets of a moderate size which are non-uniformly distributed in the space. Though a large number of distance calculations are made in each query, when compared to the k-d trees described in section 2.4, the computational overhead is much less. The reader interested in a more detailed analysis of this approach is referred to the paper of Friedman, Baskett and Shustek (4) which contains an excellent discussion of projection as applied to the nearest neighbor problem.

2.3 Cell techniques

Cell techniques are appropriate structures when the point set is constrained to be almost uniformly distributed in a subset of the Euclidean space. For example, the cell technique might be suitable to represent cities on a map of the United States, with the two dimensions in the

Euclidean space being the latitude and the longitude (in this example we ignore the spherical shape of the globe.) The cell technique structures the data by placing a "checker board" over the map and assigning each city to a square on the checkerboard. The cell in which any city belongs can be computed efficiently by truncating the latitude and longitude down to the next multiple of (say) five degrees. A cell structure can be used in higher dimensional spaces by considering points to fall in hyper-cubes. The points in the hyper-cubes can be efficiently stored as a linked list, or some other similar structure. In a sense, this structure is a multi-dimensional hashing scheme with cubical buckets. The bucket in which a particular point belongs can be determined quickly by a division operation.

Knuth has discussed this scheme for the two dimensional case in (9). Levinthal (11) used a cell technique in three dimensional Euclidean space for determining all atoms within five angstroms of every atom in a protein molecule--he referred to the technique as "cubing". Yuval applied this structure to the nearest neighbor problem in (14).

The storage required for this scheme is proportional to the number of hypercubes in the space plus the number of points. (The number of hypercubes in the space is the product of the number of regions in each of the k dimensions; for example, if 2 space is divided into 8 regions on the x-axis and 8 on the y-axis, the resulting partition is the standard checkerboard of 64 squares.) Retrieval of all near neighbors within r of point x is accomplished by examining all the points in all of the cubes within distance r of x. (The distance from a point to a cube, or any hyper-rectangular body, can be determined efficiently using the method described in section 6.) The time required for retrieval will be proportional to the sum of the number of cubes overlapped by the sphere of radius r plus the number of points within those cubes.

This technique is suitable for point sets which are uniformly distributed throughout a low dimensionality space. In a highly nonuniform space, either the cells would have to be very large (increasing search time), or much extra storage would be used. For suitable point sets, the cell structure can be implemented very efficiently in terms of both space and time.

2.4 k-d trees

The multidimensional binary search tree (in k dimensions, the k-d tree) is described by Bentley in (1). It is a generalization of the standard binary search tree as described by Knuth (10). In the standard binary search tree the decision to proceed to one of the two sons of a node is made by comparing the query key to the value of the key stored in the node. Since there are k keys associated with each point, this scheme can be extended to k dimensions by specifying in the node not only the value against which the comparison should be made, but also which dimension should be compared. For the k-d trees described in (1), every point in the space is stored in an internal node of the tree, and the structure of the tree is quite dependent on the way the points were presented to the tree building algorithm. In (5) Friedman, Bentley and Finkel discuss a modification of the k-d tree in which the points are all stored in external nodes of the tree (buckets) and the structure is determined by an "optimization prescription" which guarantees a nice partitioning of the search space. Appendix A contains a brief description of k-d trees. It is beyond the scope of this paper to describe k-d trees in detail, and the reader interested in their implementation is referred to Friedman, Bentley and Finkel (5) for such a description.

The storage required by k-d trees is proportional to n. The preprocessing time required to build a k-d tree is O(k*n*log n). The nearest neighbor search algorithm described in (5) can be easily modified to find fixed radius near neighbors. As the search algorithm visits a node it must visit one or both of that node's subtrees. The algorithm must test the bounds of both of the subtrees and visit each if and only if that subtree's bounds overlap the ball of center x and radius r. Since this is equivalent to determining if the hyper-rectangle defined by the bounds of the subtree is within distance r of point x, the test can be made efficiently using the technique described in section 6. (It might be noted for the sake of efficiency that if the bounds of the first son tested did not overlap the ball, then the bounds of the other son must overlap the ball and the test to determine so need not be made.)

The author conjectures that k-d trees provide the asymptotically optimal structure for use in the fixed radius near neighbor problems. The partitioning they impose on the space has the desirable property of conforming to the peculiarities of the data; in this sense, k-d trees are an "adaptive celling" technique. Given one dimensional data, the k-d tree will mold itself into a standard one dimensional binary search tree. Given highly uniform data, the k-d tree will impose a partitioning very similar to the cell technique described in section 2.3. Although the number of operations that must be performed to find fixed radius near neighbors is small (in proportion to the number of points), the preprocessing to construct the tree and each tree operation are relatively time consuming. For small n, the other techniques mentioned in this section might prove faster than k-d trees.

2.5 Other techniques

In this section the author has included schemes that he feels are no longer competetive with the above mentioned techniques. These are included both for historical completeness and in the hope that someone might be inspired by one of these ideas to invent a new technique for the problem.

2.5.1 Recursive cells

Knuth points out that the notion of cells can be applied recursively (9). That is, when one of the cubes has more than some certain number of points, that cube is further divided into subcubes of yet smaller size. This scheme implies a multidimensional tree with multiway branching. In terms of both the partitioning imposed on the space and the ease of implementation, this idea seems to be dominated by the quad tree (section 2.5.3), which is in turn dominated by the k-d tree (section 2.4).

2.5.2 Post office trees

A tree structure developed by Bruce McNutt for the fixed radius near neighbor problem is described by Knuth in (9). The structure has been named the "post-office tree". Each node of the tree corresponds to a point in 2 space and a "test radius". A distance calculation determines which of a node's two sons should be visited. The preprocessing of the structure is done for a fixed radius r, so r can not vary between queries. The structure uses a high redundancy of storage. For example, a tree was built that contained the 231 most populous cities in the continental United States. For a value of r = 35 miles, a tree of 1600 nodes was required. Because of the high storage redundancy, this scheme appears to be inferior to k-d trees.

2.5.3 Quad trees

Quad trees were described by Finkel and Bentley in (3). They are a generalization of the binary tree in which every node has 2**k sons. Bentley and Stanat (2) analyzed the performance of quad trees for fixed radius near neighbor searches in 2 space using the maximum coordinate metric in uniform point sets. John Linn discussed in his thesis (2) the fact that quad trees (which he called "Search-sort k trees") have advantages over binary trees when used in a synchronized multiprocessor system. This application aside, however, quad trees seem to be dominated by their historical successor, k-d trees.

2.5.4 Voronoi diagrams

Voronoi diagrams are polygonal graphs which induce a fascinating partition on point sets in 2 space. Michael Shamos has described their application in many diverse areas in (13). John Zolnowsky (15) has applied Voronoi diagrams to finding fixed radius near neighbors in 2 space for "sparse" point sets. Sparse sets have the property that no point has more than c neighbors within radius r. In such a space, Zolnowsky's algorithm requires O(n*log n) preprocessing time and has a query response time of O(log n + c). Not only are these times asymptotically efficient, but the algorithms can be implemented efficiently for point sets of practical size. Unfortunately, Voronoi diagrams have not (yet) been extended to 3 space or higher, so Zolnowsky's results hold only for 2 space.

2.5.5 Multiway cluster trees

In (6) Fukunaga and Narendra discuss a tree structure which allows the branch and bound technique of operations research to be employed in finding nearest neighbors. The algorithm uses a clustering procedure to determine the subtrees of each node. The covering induced on the search space is very irregular and includes much overlap. A comparison of these trees with k-d trees is available for the nearest neighbor problem. To find the nearest neighbor in 2 space among a thousand points required 61 distance calculations in the experiments of Fukunaga and Narendra. Similar experiments in k-d trees reported by Friedman, Bentley and Finkel (5) showed that k-d trees required only 4 distance calculations. Thus k-d trees seem to be superior.

2.5.6 Multidimensional search tries

Edward McCreight (private communication) has proposed a scheme by which the bits representing the coordinates of the points are merged together into a "superkey". That is, the first bit of the superkey for point x will be the first bit of x1, the second bit of the superkey is the first bit of x2, and so on, until the k+1st bit of the superkey is the second bit of x1, and the cycle repeats. The records are stored in a table sorted by the superkey. A fixed radius near neighbor search can avoid examining large parts of the table using this scheme. In a sense, this structure is to k-d trees as digital search tries are to standard binary trees. This scheme needs to be studied more carefully, but it appears to be inferior to k-d trees.

2.5.7 Sophisticated cell techniques

Yuval has suggested in (14) more sophisticated cell techniques than the simple "cubing" techniques described in section 2.3. Among these are a system of overlapping cubes designed such that the response to any query will be found in only one cell. This scheme requires storage redundancy exponential in dimension. He also suggests that a hexagonal covering of 2 space might be more efficient than the square covering. Though this covering would indeed reduce the number of distance calculations made, the computational overhead would be costly compared to the relatively cheap cubing scheme.

2.5.8 Distribution dependent cell techniques

It was pointed out in section 2.3 that straightforward cell techniques are inappropriate if the distribution of points in the space is highly nonuniform. If, however, the probability distribution of the points is known a priori, this information could be used to create cells which are large in the sparse regions of the space and small in the dense regions, so that every cell contained appropr cximately the same number of points. This approach would function only if the exact probability distribution of the point set was known beforehand--any deviation from that exact distribution might be disastrous. It is interesting to note that k-d trees automatically provide this distribution dependent partitioning, without being given the exact distribution.

3. The Batch Model

Oftentimes queries for all fixed radius near neighbors to points do not just trickle into the system haphazardly--they arrive in batches. Examples of this are numerous. This occurs when queries are generated by a number of users at remote terminals and collected in concentrators before they are sent (in a batch) to the central computer. As a substructure of a molecule is being rotated in 3 space, it might be desired to find all fixed radius near neighbors to all atoms in the substructure; thus the atoms in the substructure form the batch. Such batch queries could be answered by merely iterating the online techniques discussed in section 2 for each point in the batch, but there are other more sophisticated ways of handling the problem. We will assume that the radius r is the same for all points in the batch, and that the number of points in the batch (for notation) is m.

3.1 Brute force

When the points in the batch are allowed to be arbitrary points, one can not do any better than to compute the distances between each of the m points in the batch and each of the n points in the point set, for a total of m*n distance calculations. If the m points are constrained to be in the point set, then one could reduce the number of distance calculations made by using the following strategy. First, compute by brute force all of the near neighbor pairs among the m points in the batch using the technique described in section 4.1, which uses m**2/2 distance calculations. Now for each of those m points in the batch, mark their representations in the main point set as having been present in the batch (we know we have already found all of their near neighbors in the batch.) For each of the m points in the batch, compute all the distances to the n-m points not in the batch, finding all near neighbors in m*(n-m) distance calculations. Using this method, all of the near neighbor pairs will be found using only n*m - m**2/2 distance calculations, a savings of m**2/2 calculations over the naive approach. The overhead involved in implementing this approach would make it practical only if m were fairly large.

3.2 Projection

A variant of the data processing technique of sequential file updating can be used to increase the efficiency of batched fixed radius near neighbor searches. The procedure works by choosing a dimension (perhaps the one with the greatest variance in the point set) and sorting the points in the batch by that dimension. The next step is to sequence through both the batch and the point set lists in parallel, reducing the search time by noticing that points separated by r in one dimension are of distance greater than r apart in k space. There are many ways of implementing this philosophy. One might have an outer loop going through all the points in the point set and an inner loop iterating through that subset of the points in the batch that are within r (in the chosen dimension) of the current point in the point set. This technique avoids the time required for m binary searches at the cost of going through the entire list, so it is feasible only when n is less than m * log n.

3.3 Cell techniques

One possible approach to speeding up batched queries when using cells as the storage structure would be to group together all the points in the batch that fall in the same cell. Then all the points in the surrounding cells (all cells within r of the given cell) would be compared to those points from the batch. This might be particularly efficient if the points in the batch happened to be bunched rather closely together.

3.4 k-d trees

One of the most expensive aspects of searching a k-d tree is the overhead of tree traversal incurred when stacking the nodes to be revisited and in updating the bounds arrays. If a number of queries are present in a batch, it might be worth-while to perform some other bookkeeping in order to incur this overhead only once. That could be accomplished by performing one traversal of the tree, keeping track at each node of those points in the batch that are currently "active" in the traversal by use of a bitstring or some other set implementation. Each such bitstring would require m bits, and in a balanced tree the depth of the recursion stack is bounded by log n, so there would be only m*log n bits of storage required to do the bookkeeping. At each internal node of the tree, a point in the batch is considered active if and only if its corresponding bit is one. Before proceeding to one of the two subtrees the search procedure tests for every active point to see if the bounds of the subtree are within r of that point, and sets the bits of the node's son's bitstring to one if and only if the bounds are within r. If none of the bits are one, then the search does not bother to proceed down the subtree. When the search visits a bucket it compares all the points in the bucket to all the active points in the bitstring, and reports all near neighbors thus found. It might be that if the cardinality of the set shrank below a certain size, then some other set representation, such as a linked list, might be more efficient than the bitstring. This method (using bitstrings) has been implemented by the author, and for batches of size n the program ran in about half the time of the iterated use of the online strategy.

4. The All Close Pairs Problem

The all close pairs problem can be stated as follows: given a collection of n points in k space, enumerate all pairs among the n within distance r of each other. One could approach this problem using the simple online or batch models, but there are better methods that can be used. One reason for this is that those solutions would enumerate all close pairs twice: once when x was within r of y, and again when y was found to be within r of x. Avoiding this redundancy can yield up to a factor of two speedup.

4.1 Brute force

The following program (in pseudo-ALGOL) will solve the all close pairs problem using n(n-1)/2 distance calculations:

```
for i := 1 until n-1 do
    for j := i+1 until n do
        if distance(i,j) <= r then
            report <i,j> as a close pair;
```

The program is easy to implement efficiently on a computer.

4.2 Projection

The projection method can be used to search for all close pairs. The first step of an algorithm based on this technique is to choose the sparsest dimension (in some sense). This could be accomplished in O(n*k) time by calculating the variances of the points in all dimensions and choosing the dimension of maximum variance. Once a dimension is chosen, the points are sorted by that dimension, which requires O(n*log n) time. This structure can be used immediately to find the set of all close pairs (without redundancy). The outer loop of the procedure would consider the elements of the list in order, maintaining a pointer to the present element in the list and the first element in the list greater than r away from the present element in the chosen dimension. These two pointers together define a sublist. The inner loop traverses this sublist, comparing every point on it to the current point in the outer loop, and reports all near neighbors. The number of distance calculations made using this method is bounded by n(n-1)/2, and for many distributions will be substantially less. The overhead involved in this technique is small if a good sorting algorithm is used.

4.3 Cell techniques

The most simple nonredundant all pairs algorithm for a cube structure is merely to iteratively search for all points in the half sphere of radius r centered at all points (use of half spheres avoids redundancy). A more sophisticated scheme would, for each cube, compute all near neighbor pairs with one point in the given cube and the other point in any cube within r of the given cube. Note that the naive implementation of considering for every cube all other cubes within r will lead to redundancy. Therefore, a method such as that used in section 4.2 should be employed here to consider each pair of cubes only once. One implementation of this scheme would employ the knowledge of the relationship of r to the length of the edge of a cube. For instance, if the edge size was exactly r, then the larger cube of side 3*r formed by taking all the cubes adjacent to any cube C would certainly contain all of near neighbors to any point in C. This large cube will contain 3**k-1 cubes besides C. It is easy to see a scheme where one looks for near neighbors to points in C in only (3**k-1)/2 of C's neighboring cubes, and then the remainder of the pairs are found as C's neighbors investigate C. For example, in 2 space (using the compass system), any cube need consider only its N, NE, E, and SE neighbors; then it will in turn be a N neighbor to its S neighbor, and likewise for its SW, W, and NW neighbors. This scheme could be implemented very efficiently using well known techniques for multidimensional arrays (see Gries (7) for a discussion of the implementation of multidimensional arrays.)

4.4 k-d Trees

The most simple nonredundant all pairs algorithm for the k-d tree is the same as for a cube structure, that is, searching for all points in half spheres centered at all points. A more sophisticated scheme would, for each bucket, find all other buckets within r, and examine all the points in the two buckets for fixed radius near neighbors. Naively implemented this scheme would compare each bucket to every other near bucket twice, just as the naive implementation of the cell technique would. However, avoiding the redundancy is not so easy with k-d trees as it is with cells, due to the irregular shape of the buckets. One way to avoid this is to define some relation R on the cells such that (C1 R C2) if and only if not (C2 R C1), for any cells C1 and C2. Then C2 would be examined when searching for C1's near neighbors, because (C1 R C2), but C1 would not be searched when searching for C2's near neighbors. A particularly nice relation is Ci R Cj if and only if i < j. This relation is easy to test for pairs of buckets. This relation can also be used to prune the search of the tree if the buckets are numbered such that all the buckets in the left subtree of a node are numbered less than a given field of the node, and all buckets in the right subtree have a greater bucket index than the field.

5. Dynamic Point Sets

Many applications of fixed radius near neighbor searching deal with point sets that are dynamic--that is, points can be inserted, deleted, or change their locations (though a change of location could be accomplished by a deletion followed by an insertion, we will see later that change is a useful primitive.) As such events take place, it is desirable to change as little of the data structure as possible. In this section we will investigate some of the techniques used for dealing with the problem of dynamic changes to the various structures we have discussed.

The most simple type of change is when a single point is modified. When the modification is an insertion or deletion, the corresponding structure can be easily modified using the brute force, projection, or cell techniques quite easily. Insertions and deletions with k-d trees are a bit more difficult. They could be made by inserting or deleting the items in the buckets, but that is disastrous if successive insertions are going into only a few buckets--the tree would grow quite out of balance. In this case the tree should not use buckets but instead store the points in the nodes of the tree and use the dynamic insertion and deletion routines described in (1). For any of the structures, if the set of all close pairs in the point set was known before a position change of point x, the new set of all close pairs can easily be calculated after the position change of x. The only previous close pairs that are no longer close pairs are points that were close to x before x moved, and the only new close pairs are the new near neighbors to x.

Any batch of changes can be handled by iterating the above techniques. A more radical situation occurs when all the points in the structure change their locations. If the new locations are totally unrelated to the old ones, not much can be done besides throwing out the old structure and starting over again from scratch.

Oftentimes, however, the new positions are related to the old ones. Consider the case of an air traffic control system which updates the locations of the planes in the system by a radar scan every five seconds. It is safe to assume (for today's civilian aircraft) that no plane will have travelled more than a mile from its previous position in five seconds. Using such restrictions (that a point will be no more than some constant distance from its old position), it is possible to develop very fast updating algorithms. For the projection technique, the new projection will very likely closely resemble the old, and a variation of the bubble sort could be used to modify the projection. The main loop of the procedure changes the position of each point. As the position of a point is changed, it "bubbles" up or down the projection until it finds its new home. This procedure will efficiently give the new projection. For k-d trees a similar notion can be used. When the location of each point changes, test if the change is great enough to cause the point to move from its present bucket to a new bucket. If not, all is well; otherwise, transfer the node to the new bucket. Once again, this strategy runs the risk of creating unbalanced trees. One way to counter this is to use the same buckets until the tree is found to be out of balance, then reorganize the tree at that time.

6. Implementation Suggestions

In this section we will discuss some of the problems associated with implementing the schemes we have investigated. The issues raised in this section might also be important for future techniques not discussed in this paper.

6.1 Computing interpoint distances

It is easy to compute the distance from x to y by summing the squares of their differences in each dimension, and then taking the square root of the sum. In many of the schemes in this paper, however, the complete interpoint distance is not needed--we only need to know if x is greater than r from y. Since square root is a monotonically increasing function, the sum of the squares of the differences could be compared to r**2, and the costly square root operation avoided. A more sophisticated technique, appropriate for very high dimensional spaces and point sets with few near neighbors, is to compute the sum of the squares of the distances only until they exceed r**2. This involves making a comparison after each addition to the sum of the squares. This is cost effective if a few comparisons help to avoid many multiplications and additions. This idea could be used in a limited form by testing after every few additions.

6.2 Computing distances to hyper-rectangles

Using the cell and k-d tree structures, it is necessary to compute the distance from a hyper-rectangle (which could be a cell, bucket, or the bounds of a subtree) to a point. A modification of the bounds_overlap_ball procedure described by Friedman, Bentley and Finkel (5) can be used to efficiently calculate the distance. The procedure notices that the distance from a point to a hyper-rectangle is the distance from the point to the closest point in the hyper-rectangle, and that the location of that point can be deduced by considering the k dimensions independently.

The procedure calculates the distance from the point to the hyper-rectangle in each dimension and sums the squares of these distances to obtain the desired answer. The hyper-rectangle is defined by its lower and upper bounds in each dimension (it is assumed to be rectilinearly oriented in the space). The distance from a given point x to the hyper-rectangle in a given dimension (say i) is calculated as follows: if xi is between the lower and upper bounds, the distance is zero; otherwise it is the distance from xi to the closest bound. As with distances between points, often one is only interested if the distance from a point to a hyper-rectangle is less than some r. If this is all that is desired, a similar technique to that described in section 6.1 of computing the partial sum only until it exceeds r (or the total sum has been computed) can be employed.

The algorithm described here can also be used to compute distances between points and hyper-rectangles for other metrics. If the city block metric were used, then the sum of the distances (not their squares) in each dimension would be computed. If the maximum coordinate metric were used, then the maximum single dimensional distance would be recorded. The algorithm can also be used to calculate distances between two hyper-rectangles. For that application the single dimensional distance is zero if the line segments defining the two rectangles in that dimension overlap, otherwise it is the distance between the line segments.

6.3 Using multiple structures in refinement

Instead of employing only one structure, it might be effective to use different structures for different parts of the problem. In effect, the k-d trees and cell structures as described use the brute force technique in the buckets and cells in which the points are stored. It might be cost effective to sort the elements in a bucket by one dimension and use a modification of the projection algorithm to search in the bucket if it were very large. The recursive cell structure defined in section 2.5.1 is an example of refinement using the same structure. Other more exotic kinds of refinements can be imagined.

6.4 Cells within the fixed radius

In either the k-d tree or cell structures, if it is determined that the entire bucket (or cell) is within the fixed radius r of the point x, then clearly all the points in the bucket are within r of x. Whether a cell is entirely within a fixed radius ball can be determined quickly using a method similar to that described in section 6.2. It is sufficient to determine if the point in the cell furthest from x is within r. The point in the cell furthest from x is (by the independence among the dimensions for the metrics we have used) that point which maximizes the distance to xi in each dimension i. This point can easily be determined in time linear in k. Once again, as in sections 6.1 and 6.2, the partial sum of the distance squared need only be computed until it exceeds r.

Jerome Friedman has pointed out that this technique will prove extremely valuable when k-d trees are used in point sets that have sub-collections of great local density. This "bounds within ball" test could note that a whole subtree was in the region of interest. When that was noticed, the search procedure would merely traverse the subtree, enumerating all the points therein as near neighbors.

7. Areas For Further Work

Very little is known about the behavior of the structures described in this paper. The performance of these structures for different distributions of points should be carefully analyzed to determine how many distance calculations each structure needs to solve the different problems mentioned. The algorithms described should be coded efficiently in some common language (such as FORTRAN) and the details of implementation overhead carefully analyzed so that the knowledge of the number of distance calculations made can be translated into actual running times on common computers. The implementation parameters, such as bucket sizes for k-d trees and cell sizes for the cell method, need to be analyzed in detail.

The problem of points in motion discussed in section 5 is very important, especially as it relates to traffic control systems. Much more work needs to be done in this area.

ACKNOWLEDGMENT

The author gratefully acknowledges both the technical advice and the general encouragement given by Forest Baskett, Fred Brooks, Don Stanat, and Bill Wright.

Bibliography

1. Bentley, J. L. Multidimensional binary search trees used for associative searching. To appear in Communications of the ACM.

2. Bentley, J. L. and D. F. Stanat. Analysis of range searches in quad trees. To appear in Information Processing Letters.

3. Finkel, R. A. and J. I. Bentley. Quad trees: A data structure for retrieval by composite key. Acta Informatica 4(1), 1-9. 1974.

4. Friedman, J. H., F. Baskett, and L. J. Shustek. An algorithm for finding nearest neighbors. To appear in IEEE Transactions on Computers.

5. Friedman, J. H., J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic time. Stanford CS Report STAN-CS-75-482. Submitted to Transactions on Mathematical Software.

6. Fukunaga, K, and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. IEEE Transactions on Computers C-24 (7), 750-753. July, 1975.

7. Gries, D. Compiler Construction for Digital Computers. John Wiley and Sons, 1971.

8. Iverson, K. E. A Programming Language. John Wiley and Sons,

1962.

9. Knuth, D. E. The art of computer programming, volume 3, Sorting and searching. Addison-Wesley, 1973. Section 6.5.

10. -----, section 6.2.2.

11. Levinthal, C. Molecular model-building by computer. Scientific American 214, 42-52. June 1966.

12. Linn, J. General methods for parallel searching. Technical Report Number 61, Digital Systems Laboratory, Stanford Electronics Laboratory, Stanford University. May, 1973.

13. Shamos, M. I. Problem Book in Computational Geometry. To appear as a Carnegie-Mellon University Computer Science Department Report.

14. Yuval, G. Finding near neighbours in k-dimensional space. Information Processing Letters 3(4), 113-114. March, 1975.

15. Zolnowsky, J. To appear as a SLAC-PUB.

Appendix A. An introduction to k-d trees

The k-d tree is a generalization of the simple binary tree used for sorting and searching. The k-d tree is a binary tree in which each node represents a subcollection of the points in the space, and a partitioning of that subcollection. The root of the tree represents the entire cellection. Each nonterminal node has two successors. These successor nodes represent the two subcollections defined by the partitioning. The terminal nodes represent mutually exclusive small subsets of the points, which collectively form a partition of k space. These terminal subsets are called buckets.

In the case of one-dimensional searching, a point is represented by its value in a single dimension and a partition is defined by some value of that dimension. All records in a subcollection with key values less than or equal to the partition value belong to the left successor node, while those with a larger value belong to the right successor. The dimension thus becomes a discriminator for assigning records to the two subcollections.

In k space, a point is represented by k dimensions. Any one of these can serve as the discriminator for partitioning the subcollection represented by a particular node in the tree; that is, the discriminator can range from 1 to k. The original k-d tree proposed by Bentley (1) chose the discriminator for each node on the basis of its level in the tree; the discriminator for each level was obtained by cycling through the keys in crder. That is,

$D = L \bmod k + 1$

where D is the discriminator for level L and the root node is defined to be at level zero. The partition values were chosen to be random key values in each particular subcollection.

In (5) Friedman, Bentley and Finkel describe a way of constructing an "cptimized" k-d tree. Instead of cycling through the dimensicns, the discriminator for a node is that dimension which exhibits the largest variance in the subcollection. The variance can be calculated easily for each dimension, and the maximum is chosen. The partition value is chosen to be the median value in the discriminator dimension. This construction imposes a nicely shaped partitioning on the space.

Associated with each each node or bucket in a k-d tree is a set of geometric bounds within which the points in that subcollection must lie. The bounds are represented by two one dimensicnal arrays of k elements, called LCWER and UPPER. For a given subcollection S it must be true for every node x in S and every dimension i that LOWER(i) <= xi <= UPPER(i). The bounds are updated during the descent in the tree to a subcollection by replacing UPPER(i) with the partition value of an i-discriminator node when going to its left son, and modifying LOWER(i) accordingly when visiting a node's right son. The root's upper and lower bounds are set initially to plus and minus infinity, respectively. These bounds form a rectilinearly oriented hyper-rectangle in k space within which all the points in any subcollection must be.

A fixed radius near neighbor search procedure is easily defined recursively. When visiting a node x, it computes the bounds for x's right son and visits the right son recursively if and only if its bounds are within r of the target point, and likewise for the left son. When visiting a bucket, it scans all points in the bucket to see if any are within r of the target point.