

## Chapter 63

# An Optimal Algorithm for Approximate Nearest Neighbor Searching

Sunil Arya\*    David M. Mount†    Nathan S. Netanyahu‡    Ruth Silverman§  
Angela Wu¶

### Abstract

Let  $S$  denote a set of  $n$  points in  $d$ -dimensional space,  $R^d$ , and let  $\text{dist}(p, q)$  denote the distance between two points in any Minkowski metric. For any real  $\epsilon > 0$  and  $q \in R^d$ , a point  $p \in S$  is a  $(1 + \epsilon)$ -approximate nearest neighbor of  $q$  if, for all  $p' \in S$ , we have  $\text{dist}(p, q)/\text{dist}(p', q) \leq (1 + \epsilon)$ . We show how to preprocess a set of  $n$  points in  $R^d$  in  $O(n \log n)$  time and  $O(n)$  space, so that given a query point  $q \in R^d$ , and  $\epsilon > 0$ , a  $(1 + \epsilon)$ -approximate nearest neighbor of  $q$  can be computed in  $O(\log n)$  time. Constant factors depend on  $d$  and  $\epsilon$ . We show that given an integer  $k \geq 1$ ,  $(1 + \epsilon)$ -approximations to the  $k$ -nearest neighbors of  $q$  can be computed in  $O(k \log n)$  time.

### 1 Introduction.

Let  $R^d$  denote real  $d$ -dimensional space, and let  $\text{dist}(\cdot, \cdot)$  denote any Minkowski  $L_m$  distance metric.<sup>1</sup> Given a set of  $n$  points  $S$  in  $R^d$ , and given any point  $q \in R^d$ , a *nearest neighbor* to  $q$  in  $S$  is any point  $p \in S$  that minimizes  $\text{dist}(p, q)$ . Answering nearest neighbor queries is among the most important problems in computational geome-

try because of its numerous applications to areas such as data compression, pattern recognition, statistics, and learning theory.

The problem of preprocessing a set of  $n$  points  $S$  so that nearest neighbor queries can be answered efficiently has been extensively studied [2, 3, 4, 7, 8, 14, 17, 18, 20]. Nearest neighbor searching can be performed quite efficiently in relatively low dimensions. However, as the dimension  $d$  increases, either the space or time complexities increase dramatically. We take  $d$  to be a constant, independent of  $n$ . (For example, our interest has been to provide reasonably efficient algorithms for values of  $d$  ranging up to about 20.) For  $d \geq 3$ , there is no known algorithm for nearest neighbor searching that achieves both nearly linear space and polylogarithmic query time in the worst case.

The difficulty of finding an algorithm with the above performance characteristics for nearest neighbor queries suggests seeking weakened formulations of the problem. One formulation is that of computing *approximate nearest neighbors*. Given any  $\epsilon > 0$ , a  $(1 + \epsilon)$ -nearest neighbor of  $q$  is a point  $p \in S$  such that, for all  $p' \in S$

$$\frac{\text{dist}(p, q)}{\text{dist}(p', q)} \leq 1 + \epsilon.$$

Arya and Mount [2] showed that given a point set  $S$  and any  $\epsilon > 0$ , the point set can be preprocessed by a randomized algorithm running in  $O(n^2)$  expected time and  $O(n \log n)$  space, so that approximate nearest neighbor queries can be answered by a randomized algorithm that runs in  $O(\log^3 n)$  expected time.

In this paper we improve this result in a number of ways. We present an algorithm that preprocesses a set of  $n$  points in  $R^d$  in  $O(n \log n)$  time, and produces a data structure of space  $O(n)$  such that for query point  $q$  and any  $\epsilon > 0$ , approximate nearest neighbor queries can be answered in  $O(\log n)$  time. This improves the results of Arya and Mount significantly in the following respects.

- Space and query time are asymptotically optimal (for fixed  $d$  and  $\epsilon$ ) in the algebraic decision tree model of computation.

\*Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany.

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland. The support of the National Science Foundation under grant CCR-9310705 is gratefully acknowledged.

‡Space Data and Computing Division, NASA Goddard Space and Flight Center, Greenbelt, Maryland, and Center for Automation Research, University of Maryland, College Park, Maryland. This research was carried out while the author held a National Research Council NASA Goddard Associateship.

§Department of Computer Science, University of the District of Columbia, Washington, DC, and Center for Automation Research, University of Maryland, College Park, Maryland.

¶Department of Computer Science and Information Systems, The American University, Washington, DC.

<sup>1</sup>For any integer  $m \geq 1$ , the  $L_m$ -metric is defined by the following distance function on two points  $p = (p_1, \dots, p_d)$  and  $q = (q_1, \dots, q_d)$ :

$$\text{dist}_m(p, q) = \left( \sum_{1 \leq i \leq d} |p_i - q_i|^m \right)^{1/m}$$

The  $L_1$ ,  $L_2$ , and  $L_\infty$  metrics are the well-known Manhattan-, Euclidean- and max-metrics.

- The preprocessing is independent of  $\epsilon$ , so that one data structure can answer queries for all degrees of precision.
- All algorithms are deterministic, rather than randomized, but the code is still quite simple.
- Constant factors depending exponentially on dimension have been eliminated from the preprocessing time and space. (Exponential constant factors still remain in the query time.)

Because this problem is of considerable practical interest, the importance of the last item cannot be overstated. When dealing with large point sets ( $n \geq 10,000$ ) in moderately large dimensions (say  $d \geq 12$ ), constant factors in space that are on the order of  $2^d$  or  $(1/\epsilon)^d$  (even with  $O(n)$  space) are too large for practical implementation. Of course, exponential factors in query time are also undesirable, but we will see later that for many point distributions it is possible to terminate the search algorithm early and still produce results of good, albeit unproven, quality.

We begin with an outline of our data structure and search algorithm. Our methods are based largely on existing techniques with a number of straightforward adaptations for our particular problem. The preprocessing is based on the standard technique of *box-decomposition*, which has been presented in a number of roughly equivalent forms elsewhere [4, 5, 6, 19]. In this technique points are recursively subdivided into a collection of  $d$ -dimensional rectangles with sides parallel to the coordinate planes. These rectangles are used to construct a subdivision of space into *cells* each of constant complexity. We maintain the property that the cells are “fat” in the sense that the ratio of the longest to the shortest side is bounded above by a constant. Each cell is associated with one data point that is “close” to the cell, either contained within the cell or in a nearby cell. Closeness is defined relative to the size of the cell.

Assume this structure has been built. We perform approximate nearest neighbor queries by locating the cell that contains the query point  $q$ , and enumerating cells of the subdivision in increasing order of distance from the query point. Throughout we maintain the closest data point encountered so far (see Fig. 1(a)). The hierarchical and rectangular nature of the box-decomposition makes point location and enumeration easy (although some care is needed since the decomposition tree need not be balanced). Since cells are enumerated in order of increasing distance, whenever the distance to the current cell times  $(1 + \epsilon)$  exceeds the distance to the closest known point  $p$ , the search is terminated, and  $p$  is reported as an approximate nearest neighbor.

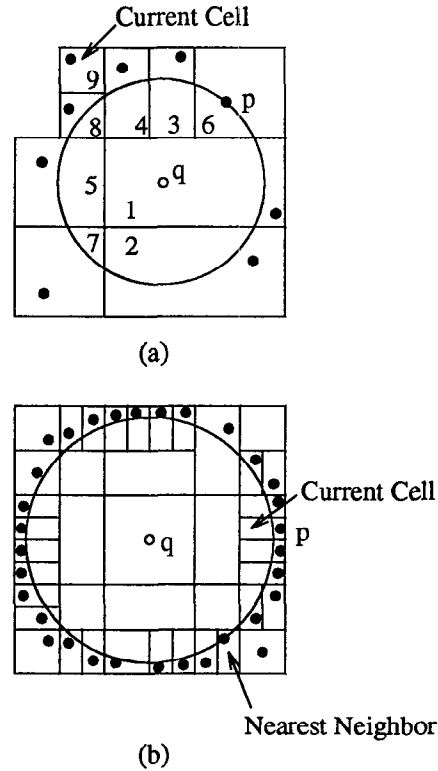


Figure 1: Algorithm overview.

How many cells might be encountered before this condition is met? We claim that the number of cells may depend on  $d$  and  $\epsilon$ , but not on  $n$ . To see this, we will show that if the search encounters a sufficiently large number of cells, depending on  $\epsilon$  and  $d$ , then it can be argued from the fatness of the cells that at least one of these cells must be sufficiently small, and hence there is a data point nearby (e.g., see Fig. 1(b)). Since no significantly closer point has been seen so far, this nearby point is an approximate nearest neighbor.

It is an easy matter to extend the algorithm for computing nearest neighbors to actually enumerating points in approximately increasing distance from the query point. In particular we show, that after the same preprocessing, given any point  $q$ ,  $\epsilon > 0$ , and  $k$ , the  $(1 + \epsilon)$ -approximate  $k$  nearest neighbors can be computed in  $O(k \log n)$  time.

Space and query time are asymptotically optimal under the algebraic decision tree model. It is easy to see that  $\Omega(n)$  space and  $\Omega(\log n)$  time are needed under this model even in dimension one, since the data structure must be able to support at least  $n$  possible outputs, one for each query that is equal to a point of the set. It should be pointed out that common techniques for the closest pair problem (e.g. [11, 13]) violate this model

by making use of the floor function to achieve efficient running times; however, it is not clear whether this observation can be applied to the approximate nearest neighbor problem.

## 2 The Data Structure.

Recall that we are given a set of  $n$  data points  $S$  in  $R^d$ . We will use the term *data point* when referring to points in  $S$ , and *point* for arbitrary points in space. As mentioned earlier the data structure is a straightforward adaptation of the standard *box-decomposition* construction (described below). This technique is the basis of the data structures described by Callahan and Kosaraju [5], Clarkson [6], Vaidya [19], Samet [15], Bern [4], Feder and Greene [9], and others.

Before describing the specifics of our implementation, we present a list of properties which suffice to apply our algorithms. Define a *subdivision* of  $R^d$  to be a finite collection of  $d$ -dimensional *cells* whose interiors are pairwise disjoint, and whose closures cover the space. The data structure consists principally of a subdivision of  $R^d$  into  $O(n)$  cells, each of constant complexity. We do not require that this subdivision satisfies any particular topological or geometric requirements other than those listed below (e.g. it need not be a cell complex, and cells need not be convex).

- (a) **Bounded occupancy:** Each cell contains from zero up to some constant number of data points. Points that lie on the boundary between two or more cells are assumed to be assigned to one of the cells by some tie-breaking rule.
- (b) **Existence of a close data point:** Each cell  $c$  is associated with a given positive real *size*, denoted  $size(c)$ . If  $s$  is the size of a cell, then for any point  $p$  within the cell, there exists a data point whose distance from  $p$  is at most some constant factor  $\alpha$  times  $s$ . The value of  $\alpha$  will generally be a function of the dimension. A pointer to such a data point is associated with the cell. At most a constant number of cells are associated with any one data point.
- (c) **Packing constraint:** The number of cells of size at least  $s$  that intersect an open ball of radius  $r > 0$  is bounded above by a function of  $r/s$ , independent of  $n$ . (By *ball* we mean the locus of points that are within distance  $r$  of some point in  $R^d$  according to the distance metric.)
- (d) **Point location:** Given a point  $q$  in  $R^d$ , the cell containing  $q$  can be computed in  $O(\log n)$  time.
- (e) **Distance enumeration of cells:** Define the *distance* between a point  $q$  and a cell  $c$  to be the

closest distance between the point and any part of the cell. Given  $q$ , the cells of the subdivision can be enumerated in order of increasing distance from  $q$ . The time to enumerate the nearest  $m$  cells is  $O(m \log n)$ . Note that  $m$  may not be known when the enumeration begins.

**2.1 Box-decomposition tree.** We now describe how to adapt the box-decomposition method to satisfy these properties. We call the resulting structure a *box-decomposition tree*. Care has been taken in this definition to avoid exponential factors in dimension from entering preprocessing and space bounds. The specific details of the construction are discussed in later sections.

The points of  $S$  are assumed to be represented in the standard way as a  $d$ -tuple of floating point or integer values. By a *rectangle* in  $R^d$  we mean the  $d$ -fold product of closed intervals on the coordinate axes. We will limit consideration to a restricted set of rectangles, which we call *boxes*. A *box* is a rectangle such that the ratio of its longest side to its shortest side is bounded above by some constant  $\rho$  that is greater than or equal to 2. Let us assume for simplicity that all the data points have been scaled to lie within a  $d$ -dimensional unit hypercube  $U$ .

For our purposes, a *cell* is either a box or the set theoretic difference between two boxes, one enclosed within the other. Cells of the former type are called *box cells* and cells of the latter type are called *doughnut cells*. Each doughnut cell is represented by its *outer box* and *inner box*. It is important to note the difference between boxes and cells. Boxes are used to define cells, and cells are the basic elements that will make up the subdivision.

Consider two boxes  $b$  and  $b'$ , where  $b'$  is enclosed within  $b$ . We say that  $b'$  is *sticky* for  $b$  if for each of the  $2d$  sides of  $b'$ , the distance from this side to the corresponding side  $b$  is either zero, or is greater than or equal to the width of  $b'$  along this dimension. Define  $shrink(b)$  to be a minimal, sticky box  $b'$  (different from  $b$ ) that is enclosed within  $b$  and contains the points  $b \cap S$ . Stickiness is needed for technical reasons later in Lemma 2.3.

Adapting the definitions given by Callahan and Kosaraju [5], define a *split* of a rectangle to be a partition of the rectangle into two rectangles by a hyperplane parallel to one of the coordinate axes. Define a *fair-split* of box  $b$ ,  $split(b)$ , to be a split of  $b$  into two boxes, each of which contains at least one point of  $S$ . The "fairness" of the split refers to the fact that the resulting rectangles cannot be arbitrarily thin, due to the ratio restriction on box side lengths. Henceforth, assume that all splits are fair-splits, unless stated otherwise.

LEMMA 2.1. *Given any box  $b$  which contains two or more points of  $S$ , at least one of the operations shrink and split can be performed. Once a shrink operation has been performed, a split is always possible on the resulting shrunken box.*

Box-decomposition works by starting with the unit hypercube,  $U$ , and recursively applying the operation shrink followed by split. These operations are repeated as long as the number of data points in the current box exceeds some threshold,  $BucketSize$ . The operation shrink is performed only if it is needed (since for practical reasons its overhead is much greater). A pseudocode description is given in Fig. 2. Note that a direct implementation of this procedure is not asymptotically efficient. Details of the construction are given later. The initial call is to  $BoxDecomp(S, U)$ . A set of cells will be created in the process. Each cell is represented by the rectangle(s) that define it. We make the general position assumption that data points do not lie on boundaries of cells, but this restriction is easily removed through the use of any rule for breaking ties.

```

BoxDecomp( $T, b$ ) {
  if ( $|T| \leq BucketSize$ )
    create box cell  $b$ ;
  else {
    if ( $split(b)$  is not possible) {
       $b' = shrink(b)$ ;
      create doughnut cell  $b - b'$ ;
       $b = b'$ ;
    }
    ( $b_1, b_2$ ) =  $split(b)$ ;
    BoxDecomp( $T \cap b_1, b_1$ );
    BoxDecomp( $T \cap b_2, b_2$ );
  }
}

```

Figure 2: Box-decomposition algorithm.

The decomposition is illustrated in Fig. 3, where a ratio of 2:1 is maintained for all boxes. In (a) we show the decomposition (line segments representing splits and rectangles representing shrinks) and in (b) the cells of the resulting subdivision are shown.

We can associate a binary tree with the box-decomposition in a natural way. Following Vaidya's notation, when splitting is performed we call  $b_1$  and  $b_2$  the *successors* of  $b$ . When shrinking is performed  $b'$  and  $b - b'$  are the successors of  $b$ . The successors define a binary tree, whose root is the initial hypercube  $U$ , and whose leaves are either boxes that contain a single point, or doughnut cells that contain no point. Every internal node of this tree is associated with a box, and

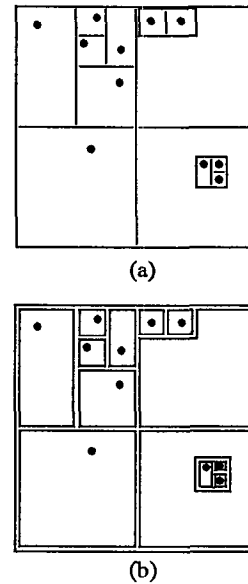


Figure 3: Box-decomposition.

this box is the disjoint union of the cells of its descendent leaves. An internal node is called a *shrinking node* if its successors arise by shrinking, and it is called a *splitting node* otherwise.

Along any path from the root to a leaf in this tree, there can be no two consecutive shrinking nodes. Thus the number of splitting nodes is at least half the total number of internal nodes. Since each split gives rise to a nontrivial partition of  $S$ , the number of splitting nodes is  $O(n)$ , and hence the total size of the tree is  $O(n)$ . Each node of the tree is associated with its bounding box, or its two defining boxes in the case of the doughnuts. In this way we do not require separate storage for the cells of the subdivision since they are just the set of leaves in the tree. Observe that it is possible in principle to subdivide the hypercube using only split operations (if we were willing to allow cells containing no point of  $S$ ). The purpose of shrinking is to guarantee that the data structure will be of size  $O(n)$ .

**2.2 Decomposition properties.** In this section we will show that properties (a)-(e) hold, or will hold after appropriate augmentation to the data structure described previously. Clearly (a) holds since each cell contains at most  $BucketSize$  points. To establish (b), define the *size* of a cell to be the length of its longest side. (For a doughnut cell we define its size to be the size of the outer box.) Property (b) is established in the following lemma. Proofs has been omitted from this version of the paper. They appear in the full version of the paper.

**LEMMA 2.2.** (Existence of a close point) *Given a box-decomposition tree for a set of data points  $S$  lying in a unit  $d$ -dimensional hypercube  $U$ , and given any point  $q \in U$ , let  $s$  be the size of the subdivision cell containing  $q$ . Then there exists a point  $p \in S$  whose distance from  $q$  is at most  $s \cdot d$ . Such points can be assigned to cells so that no point is assigned to more than two cells.*

The following lemma establishes property (c). Note that there is some subtlety in proving this lemma, since it does not generally hold for box-decompositions based on arbitrary shrinking and splitting. This is the reason we introduced the stickiness property when defining shrinking. The proof has been omitted.

**LEMMA 2.3.** (Packing Constraint) *Given a box-decomposition tree for a set of data points  $S$  lying in a unit  $d$ -dimensional hypercube  $U$ , and given any point  $q \in U$ , let  $s$  be any positive real. The number of subdivision cells of size at least  $s$  that intersect a ball of radius  $r$  centered at  $q$  is on the order of  $(r/s)^d$ .*

**2.3 Point Location.** In order to establish property (d) we need to establish balance in the tree. Following Bern [4] or Schwarz, Smid and Snoeyink [16], we do this using the standard technique of centroid decomposition. (See Clarkson [6] for an alternative randomized approach.) Let us think of the box-decomposition tree as an unrooted free-tree in which the degree of each node is at most three. For our purposes define a *centroid edge* in a binary tree of  $n$  nodes to be an edge whose removal partitions the tree into two subtrees each with at most  $\lceil 2n/3 \rceil$  leaves. Taking centroids, the nodes of the box-decomposition tree can be recursively restructured into a binary tree of  $O(\log n)$  depth. The centroid decomposition tree can be computed in  $O(n)$  time (see e.g. [12]). We assume that the resulting *centroid decomposition tree* is a second tree threaded through the same set of nodes as the box-decomposition tree.

To perform point location, consider the removal of a centroid edge  $(x, y)$ , where  $x$  is a parent of  $y$  in the box-decomposition tree. The node  $y$  is associated with either a bounding box or is a leaf corresponding to a doughnut cell in the box-decomposition tree. In the latter case, on the order of  $d$  comparisons suffice to determine whether  $q$  is contained within the doughnut. In the former case, the cells associated with  $y$ 's subtree lie within the bounding box associated with  $y$ . All of the cells associated with  $x$  lie outside of this box. Note that because of the removal of prior centroids, the region associated with  $y$ 's remaining subtree or  $x$ 's remaining subtree may be of unbounded complexity. The important thing is that the box decomposition tree gives us a separator of constant  $O(d)$  complexity. Thus in constant time we can determine the subtree

within which to continue the search. It follows that we can determine the cell containing an arbitrary query point in  $O(\log n)$  time (with a constant factor of  $d$ ).

**2.4 Distance Enumeration.** The last remaining property to consider, (e), is how to enumerate the cells in increasing order of distance about some point  $q$ . The method we employ is called *priority search* after a similar search technique used for  $k$ - $d$  trees by Arya and Mount [1]. The idea is to store a subset of cells in a priority queue ordered by their distance from  $q$ . Recall that the distance between  $q$  and a cell is the closest distance between  $q$  and any point in the cell. Whenever a cell is removed from the priority queue we enqueue a constant number ( $O(d)$ ) of its neighboring cells. Although, in general, a cell in the box-decomposition may have up to  $O(n)$  neighbors, we will show that it suffices to consider only a constant number of neighboring cells.

Every cell in the subdivision is bounded by at most  $2d$  facets of dimension  $d - 1$ . The facets of rectangular cells are just  $(d - 1)$ -dimensional rectangles. The facets of doughnut cells are in general the difference between two  $(d - 1)$ -dimensional facets, in the special case where the inner rectangle shares a boundary with its outer rectangle. In either case, the complexity of a facet is at most  $2(d - 1)$ .

We say that two cells are *neighbors* if they share a common  $(d - 1)$ -dimensional boundary. Note that the number of neighbors is not bounded by the number of facets because the subdivision is not a cell complex. Let  $c$  be the cell containing  $q$ , and let  $f$  be any facet of  $c$ . Because of the simple structure of facets, in constant  $O(d)$  time we can compute the nearest point  $q'$  to  $q$  on (the closure of) this facet. Using point location we can determine the neighbor of  $c$  along this facet that contains the point  $q'$  and lies on the opposite side of  $f$  from  $c$ . Assuming that  $q$  is in general position, this neighboring cell is unique. (In fact it can be shown that because facets are rectangles aligned with the coordinate axes, the choice of  $q'$  is independent of which Minkowski metric is used.) Let  $neigh(c, f, q)$  denote this closest of  $c$ 's neighboring cells to  $q$  along  $c$ 's facet  $f$ . Note that this process need only be performed for facets along which the interior of the cell is closer to  $q$  than the exterior. Fig. 4 shows the neighboring cells selected by this process.

To begin the enumeration, each cell is unmarked. Begin by locating the cell that contains  $q$  in  $O(\log n)$  time using point location. Insert this cell into the priority queue and mark it. Repeatedly remove the cell  $c$  from the priority queue with the smallest distance from  $q$ . For each of the facets  $f$  of  $c$ ,  $neigh(c, f, q)$  can be

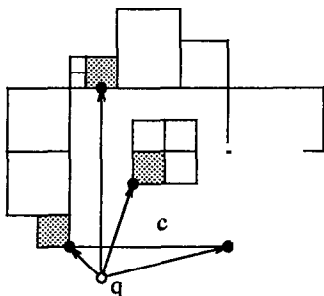


Figure 4: Distance enumeration.

computed in  $O(\log n)$  time through point location. For each such cell that is unmarked, compute its distance from  $q$  and insert the cell into the priority queue. The time needed to process a given cell is  $O(\log n)$  (with a constant factor on the order of  $d^2$ , since each of at most  $2d$  facets generate a point location query taking  $O(d \log n)$  time.)

To establish the correctness of this algorithm, observe that had we enqueued all of the neighbors of a given cell, the method would have certainly enumerated all cells in order of increasing distance. The only issue is whether by enqueueing only the closest neighboring cell for each facet no cell is missed. The following lemma establishes that for each cell  $c'$  of the subdivision, there is some cell  $c$  that lies closer to  $q$ , and some facet  $f$  of  $c$ , such that  $c' = \text{neigh}(c, f, q)$ . From this it follows that every cell will be enqueued eventually, and in proper distance order. The proof has been omitted from this version.

**LEMMA 2.4.** *Priority search visits the cells of the subdivision in increasing order of distance from the query point.*

**2.5 Construction.** We show that the box-decomposition tree can be constructed in  $O(n \log n)$  time. A naive implementation of the algorithm presented above leads to an  $O(n^2)$  time algorithm, because we have no guarantee that successive splits will be balanced. Callahan and Kosaraju [5] offer a particularly elegant solution to the problem of partitioning points, which we outline here. When we are determining how to partition the points within some box  $b$ , the data points contained in  $b$  are stored in  $d$  separate lists, each sorted by one of the coordinates, that are crossed referenced with the set of points. Rather than updating these lists after each split, a sequence of splits is performed, until each of the resulting subsets contains fewer than half of the initial number of points. In linear time it is possible to partition the initial set of points among these subsets, form the resulting sorted lists in each case, and

then recurse on each subset. Because the subproblems are all of less than half the original problem size, the overall running time is  $O(n \log n)$ . The constant factors are linear in  $d$ .

One other issue that needs to be considered is whether to shrink or to split. Let  $b$  be the current box, and let  $S'$  denote the points of  $S$  lying within  $b$ . Let  $r$  denote the smallest bounding rectangle for  $S'$ . It is an easy matter to determine from  $r$  and  $b$  whether a split is possible in constant ( $O(d)$ ) time. If no split is possible, then as we noted earlier, a shrink operation is possible. Using the length of the longest side of  $r$ , and given the ratio bound on side lengths of boxes, determine the minimum lengths of the remaining sides of any enclosing box. It is easy to see that these lengths will be less than or equal to the corresponding lengths of  $b$  (for otherwise, the ratio bound would have been violated for  $b$ ). For each dimension, enlarge the length of bounding rectangle in this dimension, while staying within  $b$ . If this is not possible for this side without violating the stickiness property, then enlarge this side's length by pushing a side of  $r$  out to the corresponding side of  $b$ . Now this side of  $r$  coincides with a side of  $b$ . In the process of enlarging, new longest side may be created, and hence side lengths that were already adjusted may need to be reevaluated. However, once a side is pushed to  $b$ 's boundary, it cannot be moved again, so after  $O(d)$  iterations, this process will terminate. The total time for this operation is  $O(d^2)$ , a constant in fixed dimensions.

**3 Approximate Nearest Neighbor Queries.**

Given a set of  $n$  points  $S$  in  $R^d$ , and assuming a data structure satisfying properties (a)-(e) of the previous section has been computed, we show how to answer an approximate nearest neighbor query in  $O(\log n)$  time (constants depending on  $d$  and  $\epsilon$ ). Let  $q$  be the query point in  $R^d$ . Recall that the output of our algorithm is a data point  $p$ , such that, for all  $p' \in S$ ,

$$\frac{\text{dist}(p, q)}{\text{dist}(p', q)} \leq 1 + \epsilon.$$

We assume that  $q$  lies within the enclosing box for the data set, but it is easy to modify the algorithm to handle the general case.

We begin by applying the point location algorithm to determine the cell containing the query point  $q$ . Enumerate the cells of the subdivision in increasing order of distance from  $q$ . Recall from (a) and (b) that each cell is associated with one or more points, either because the cell contains data points, or because it is associated with a nearby point. As each cell is visited, process it by computing the distances of these points

to  $q$  and recording the closest seen so far. The cell enumeration terminates if the current cell's distance to  $q$  times  $(1+\epsilon)$  exceeds the nearest distance  $r$  to a known data point  $p$ . We know that no subsequent point to be encountered will be closer to  $q$  than  $r/(1+\epsilon)$ , and hence  $p$  is an approximate nearest neighbor.

The processing time for each cell visited by the algorithm is  $O(1)$  with a constant factor on the order of  $d$  for distance computation. The time is dominated by the  $O(\log n)$  overhead needed at each step of the cell enumeration. We can visit the  $k$  nearest cells to  $q$  in  $O(k \log n)$  time. To establish the  $O(\log n)$  running time, it suffices to prove that the number of cells visited is  $O(1)$ .

**LEMMA 3.1.** *The approximate nearest neighbor algorithm terminates after visiting at most  $O(1)$  cells (with constant factors depending on  $d$  and  $\epsilon$  on the order of  $O((d(1+1/\epsilon))^d)$ ).*

*Proof.* The proof is based on computing an upper bound on the distance to the nearest neighbor and showing that within this distance the presence of any sufficiently small sized cell will cause the algorithm to terminate. From this we can apply property (c) from the previous section to infer that there are only a constant number of cells of a given size. Details have been omitted due to space limitations.  $\square$

#### 4 Approximate $k$ -Nearest Neighbors.

In this section we will describe a generalization of the approximate nearest neighbor procedure to the problem of computing approximations to the  $k$  nearest neighbors of a query point. A point  $p$  is a  $(1+\epsilon)$ -approximate  $k$ -th nearest neighbor to a point  $q$  if the ratio of the distance between  $q$  and  $p$  and the distance between  $q$  and its true  $k$ -th nearest neighbor is at most  $(1+\epsilon)$  (and in fact,  $p$  may lie closer to  $q$  than the true  $k$ -th nearest neighbor). By an answer to the approximate  $k$ -nearest neighbor problem we mean a list of *distinct* data points  $p_1, p_2, \dots, p_k$ , such that  $p_j$  is a  $(1+\epsilon)$ -approximation to the  $j$ -th nearest neighbor of  $q$ , where  $1 \leq j \leq k$ .

The algorithm is a simple generalization of the nearest neighbor algorithm. Iterate the nearest neighbor algorithm  $k$  times, with the following modifications. First, rather than maintaining the single closest data point to  $q$  so far, maintain the  $k$  closest points seen so far in a priority queue. Second, the termination condition presented in the previous algorithm does not cause termination, but results in the generation of a new approximate nearest neighbor taken from the top of the priority queue. Finally, some care needs to be taken to be sure that the same point is not reported twice. The total running time is  $O(k \log n)$ . Details have been omitted

from this version.

#### 5 Experimental Results.

In order to establish the practical value of our algorithms, we implemented them and ran a number of experiments. As is often the case with theoretical algorithm design, many of the features of the data structures and algorithms are included to handle certain worst-case situations that rarely arise in practice. Unfortunately, these features come at the expense of additional overhead. In our case, the features which we have chosen to omit affect the size and depth of the data structure, which in turn affect the running time of the algorithm but not its correctness. Furthermore, the effect of omission can be measured at the completion of preprocessing time. In general, after preprocessing is complete, it is easy to check the size and depth of the data structure, to determine whether more sophisticated preprocessing is warranted.

The first simplification is that no shrinking operations are performed (all decomposition is by splitting). The second is that centroid decomposition is not used to balance the resulting tree. The reason to avoid shrinking is that to determine whether a point lies within a shrunken box requires  $2d$  comparisons, in contrast with splitting in which a single comparison is needed. Consequently, in dimension 16, nodes that involve shrinking incur a constant factor of 32 times that needed for splitting. In general, shrinking should be avoided except in those situations where a very large number of trivial splits would be generated otherwise. (We never observed such situations in our experiments.) Centroid decomposition is avoided for the same reason. The price one pays for centroid decomposition is that each step of the point location processing reduces to determining whether a point lies within a given box. This requires  $2d$  comparisons, in contrast with the naive search algorithm that makes only one comparison for each corresponding step. The benefit of centroid decomposition is to guarantee that the tree is of logarithmic height. However, in our experiments the height of our trees did not exceed  $\lceil \log_2 n \rceil$  except by a small constant factor.

A further advantage of these simplifications is that the resulting data structures have essentially the same structure as a  $k$ - $d$  tree. Arya and Mount [1] suggested a nice trick for speeding up priority search in  $k$ - $d$  trees, which can be applied to the search structures presented here. In particular, because the splitting planes are orthogonal to the coordinate axes, and Minkowski metrics are used, it is possible to update the distance from each cell to the query point as we walk around the box-decomposition tree in  $O(1)$  time, rather than the straightforward  $O(d)$  time.

Our description of the data structure omitted details in the choice of splitting planes. We experimented with two schemes, which we describe below. Given a subset of the data points, define the *spread* of these points along some dimension to be the difference in the maximum and minimum coordinates in this dimension.

**Fair-split rule:** Given a box, determine the sides that can be split without violating a ratio of 3:1 between the longest and shortest sides of any box. Among these dimensions select the dimension along which the points have maximum spread, and split along this dimension. The choice of splitting points is the one that most evenly distributes points on either side of the splitting hyperplane, subject to the 3:1 ratio bound. (This rule was inspired by the splitting rule given by Callahan and Kosaraju [5].)

**Midpoint-split rule:** Given a box, consider its longest sides. Among these sides, select the one along which the points have maximum spread. Split this side at its midpoint. (This rule is an adaptation of quadtree-like decomposition [15].)

We ran experiments on these two data structures, and for additional comparison we also implemented an optimized  $k$ - $d$  tree [10]. The cut planes were placed at the median, orthogonal to the coordinate axis with maximum spread. This data structure is quite similar to simplifications described above except that there is no ratio bound on the side lengths of the resulting cells (and indeed ratios in the range from 10:1 to 20:1 and even higher are quite common). Although the  $k$ - $d$  tree is known to provide  $O(\log n)$  query time in the expected case for a special class of distributions, there are no proven worst case bounds. We know of no other work suggesting the use of a  $k$ - $d$  tree for approximate nearest neighbor queries, but the same termination given in Section 3 can be applied here. Unlike the box-decomposition tree, we cannot prove upper bounds on the execution time of query processing. Given the similarity to our own data structure, one would expect that running times would be similar for typical point distributions, and indeed our experiments bear this out.

Our experience shows that adjusting the bucket size, that is, the maximum number of points allowed before splitting, affects the running time. For the more flexible  $k$ - $d$  tree and the fair-split rule, we selected a bucket size of 5, but found that for the more restricted midpoint-split rule, a bucket size of 8 produced somewhat better results.

The following is a list of the distributions we considered. To model the types of point distributions seen in speech processing applications, the last two point distributions were formed by grouping the output of

autoregressive sources into vectors of length  $d$ . An autoregressive source uses the following recurrence to generate successive outputs:

$$X_n = \rho X_{n-1} + W_n,$$

where  $W_n$  is a sequence of zero mean independent, identically distributed random variables. The correlation coefficient  $\rho$  was taken as 0.9 for our experiments. Each point was generated by selecting the first component from the corresponding uncorrelated distribution (either Gaussian or Laplacian) and the remaining components were generated by the equation above. See Arya and Mount [1] for more information.

**Uniform:** Each coordinate was chosen uniformly from the interval  $[0, 1]$ .

**Gaussian:** Each coordinate was chosen from the Gaussian distribution with zero mean and unit variance.

**ClusNorm:** Ten points were chosen from the uniform distribution and a Gaussian distribution with standard deviation 0.05 put at each.

**Laplace:** Each coordinate was chosen from the Laplacian distribution with zero mean and unit variance.

**Correlated Gaussian:**  $W_n$  was chosen so that the marginal density of  $X_n$  is Gaussian with variance unity.

**Correlated Laplacian:**  $W_n$  was chosen so that the marginal density of  $X_n$  is Laplacian with variance unity.

Due to space limitations we only show results for two extreme cases, the uniform and correlated Laplacian. The results for other distributions are comparable.

Each experiment consisted of 100,000 data points in dimension 16 and the timing averages were computed over 1,000 query points, generated from the same distribution. In each experiment we recorded a number of statistics. In this section we present a number of statistics, which we think are relevant. The first is the number of floating point operations (i.e., any computation involving the coordinates of the points) performed by the algorithm. We feel this is a good machine-independent measure of the algorithm's running time, because it accurately includes the overhead for distance calculations, manipulation of the heap, and point location. We ran experiments for values of  $\epsilon$  ranging from 0 (exact nearest neighbor) up to 10. The results are shown in Figs. 5 and 6. Note that the scale is logarithmic.

To get a feel for the algorithm's actual performance, we computed the true nearest neighbor off-line, and



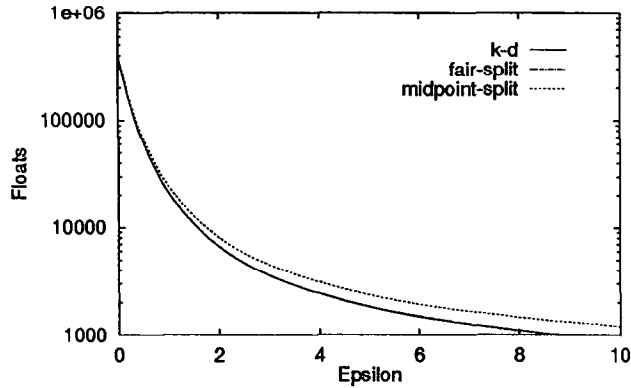


Figure 5: Uniform: Average Floating Point Operations vs. Epsilon.

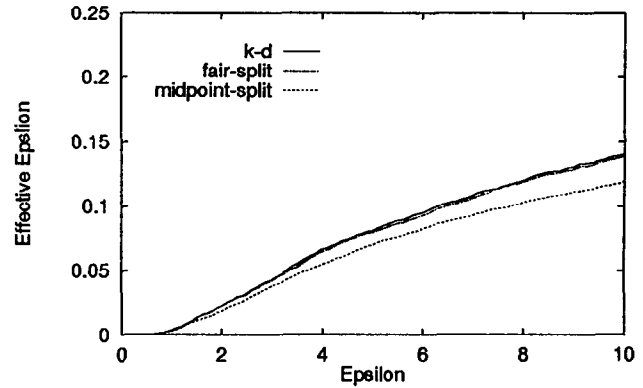


Figure 7: Uniform: Effective Epsilon vs. Epsilon.

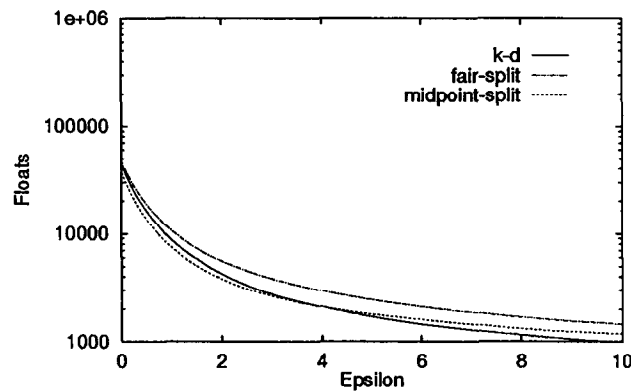


Figure 6: Correlated Laplacian: Average Floating Point Operations vs. Epsilon.

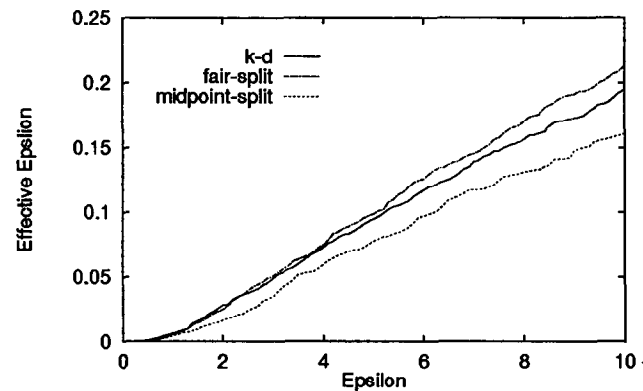


Figure 8: Correlated Laplacian: Effective Epsilon vs. Epsilon.

computed the ratio between the distance to the point reported by the algorithm and the true nearest neighbor. The resulting quantity, averaged over all query points is called the *effective epsilon*. These are shown in Figs. 7 and 8 for the same distributions.

The algorithm manages to locate the true nearest neighbor in a surprisingly large number of instances. To show this, we plotted the probability that the algorithm fails to return the true nearest neighbor for these distributions. Results are shown in Figs. 9 and 10.

The following conclusions can be drawn from these experiments.

- The algorithm’s actual performance was much better than predicted by the value of  $\epsilon$ . Even for  $\epsilon$  as high as 3 (implying that a relative error of 400% is tolerated) the effective relative error was less than 1%, and the true nearest neighbor is found almost half of the time.
- In moderately high dimensions, significant savings in running time can be achieved by computing

approximate nearest neighbors. For the  $\epsilon = 3$  cases, improvements in running time on the order of 10 to 50 were common over the exact case.

- There was relatively little difference in running time and effective performance between different splitting rules, even for the *k-d* tree, for which upper bounds on search time cannot be proved.
- For well-behaved data distributions, shrinking and centroid decomposition do not seem to be merited given the relatively high overheads they incur in running time.

## 6 Acknowledgements.

We would like to thank Michiel Smid for his helpful comments.

## References

[1] S. Arya and D. M. Mount. Algorithms for fast vector quantization. In J. A. Storer and M. Cohn, editors,

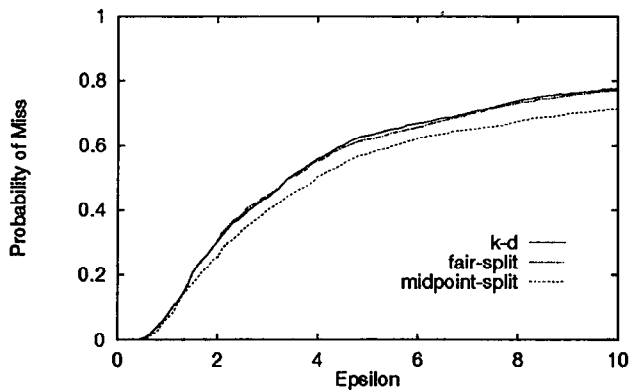


Figure 9: Uniform: Probability of Miss vs. Epsilon.

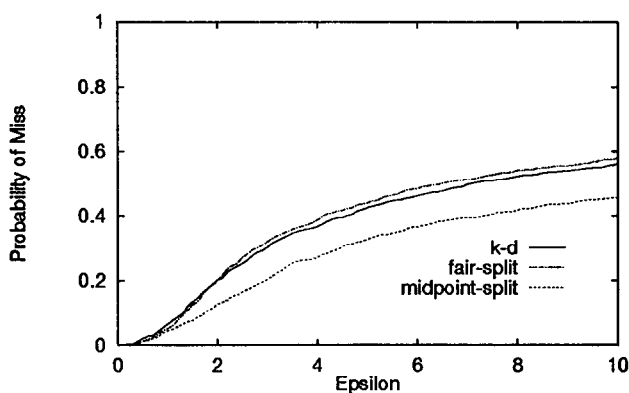


Figure 10: Correlated Laplacian: Probability of Miss vs. Epsilon.

- Proc. of DCC '93: Data Compression Conference*, pages 381–390. IEEE Press, 1993.
- [2] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.
  - [3] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected-time algorithms for closest point problems. *ACM Transactions on Mathematical Software*, 6(4):563–580, 1980.
  - [4] M. Bern. Approximate closest-point queries in high dimensions. *Inform. Process. Lett.*, 45:95–99, 1993.
  - [5] P. B. Callahan and S. R. Kosaraju. A decomposition of multi-dimensional point-sets with applications to  $k$ -nearest-neighbors and  $n$ -body potential fields. In *Proc. 24th Ann. ACM Sympos. Theory Comput.*, pages 546–556, 1992.
  - [6] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Proc. 24th Ann. IEEE Sympos. on the Found. Comput. Sci.*, pages 226–232, 1983.
  - [7] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM Journal on Computing*, 17(4):830–847, 1988.

- [8] J. G. Cleary. Analysis of an algorithm for finding nearest neighbors in euclidean space. *ACM Transactions on Mathematical Software*, 5(2):183–192, 1979.
- [9] T. Feder and D. H. Greene. Optimal algorithms for clustering. In *Proc. 20th Annu. ACM Sympos. Theory Comput.*, pages 434–444, 1988.
- [10] J. H. Friedman, J. L. Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.
- [11] M. Golin, R. Raman, C. Schwarz, and M. Smid. Randomized data structures for the dynamic closest-pair problem. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 301–310, 1993.
- [12] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear time algorithms for visibility and shortest path problems inside simple polygons. In *Proc. 2nd Annu. ACM Sympos. Comput. Geom.*, pages 1–13, 1986.
- [13] H.-P. Lenhof and M. Smid. Enumerating the  $k$  closest pairs optimally. In *Proc. 33rd Ann. IEEE Sympos. Found. Comput. Sci.*, pages 380–386, 1992.
- [14] R. L. Rivest. On the optimality of Elias's algorithm for performing best-match searches. In *Information Processing*, pages 678–681. North Holland Publishing Company, 1974.
- [15] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, Reading, MA, 1990.
- [16] C. Schwarz, M. Smid, and J. Snoeyink. An optimal algorithm for the on-line closest-pair problem. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 330–336, 1992.
- [17] M. R. Soleymani and S. D. Morgera. An efficient nearest neighbor search method. *IEEE Transactions on Communications*, 35(6):677–679, 1987.
- [18] R. L. Sproull. Refinements to nearest-neighbor searching in  $k$ -dimensional trees. *Algorithmica*, 6, 1991.
- [19] P. M. Vaidya. An  $O(n \log n)$  algorithm for the all-nearest-neighbors problem. *Discrete Comput. Geom.*, 4:101–115, 1989.
- [20] A.C. Yao and F.F. Yao. A general approach to  $d$ -dimensional geometric queries. In *Proc. 17th Ann. ACM Sympos. Theory Comput.*, pages 163–168, 1985.