Approximate Range Searching

Sunil Arya* David M. Mount†

Abstract

The range searching problem is a fundamental problem in computational geometry, with numerous important applications. Most research has focused on solving this problem exactly, but lower bounds show that if linear space is assumed, the problem cannot be solved in polylogarithmic time, except for the case of orthogonal ranges. In this paper we show that if one is willing to allow approximate ranges, then it is possible to do much better. In particular, given a bounded range Q of diameter s and $\epsilon > 0$, an approximate range query treats the range as a fuzzy object, meaning that points lying within distance ϵs of the boundary of Q either may or not be counted. We show that in any fixed dimension d, a set of n points in R^d can be preprocessed in $O(n \log n)$ time and $O(n)$ space, such that approximate queries can be answered in $O(\log n + (1/\epsilon)^d)$ time. The only assumption we make about ranges is that the intersection of a range and a d-dimensional cube can be answered in constant time (depending on dimension). For convex ranges, we tighten this to $O(\log n + (1/\epsilon)^{d-1})$ time. We also present a lower bound argument for approximate range searching based on partition trees of $\Omega(\log n + (1/\epsilon)^{d-1})$, which implies optimality for convex ranges. Finally we give empirical evidence showing that allowing small relative errors can significantly improve query execution times.

*Max-Planck-Institut für Informatik, D-66123 Saarbrücken, Germany. Email: arya@mpi-sb.mpg.de. This author was supported by the ESPRIT Basic Research Actions Program, under contract No. 7141 (project ALCOM II).

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. Email: mount@cs.umd.edu. This author was supported by the National Science Foundation under grant CCR-9310705.

1 Introduction.

The range searching problem is among the fundamental problems in computational geometry. A set P of n data points is given in d-dimensional real space, R^d, and a space of possible *ranges* is considered (e.g. d-dimensional rectangles, spheres, halfspaces, or simplices). The goal is to preprocess the points so that, given any query range Q, the points in $P \cap Q$ can be counted or reported efficiently. More generally, one may assume that the points have been assigned weights, and the problem is to compute the accumulated weight of the points in $P \cap Q$, *weight*$(P \cap Q)$, under some commutative semigroup.

There is a rich literature on this problem. In this paper we consider the weighted counting version of the problem. We are interested in applications in which the number of data points is sufficiently large that one is limited to using only linear or roughly linear space in solving the problem. For orthogonal ranges, it is well known that range trees can be applied to solve the problem in $O(\log^{d-1} n)$ time with $O(n \log^{d-1} n)$ space (see e.g. [12]). Chazelle and Welzl [7] showed that triangular range queries can be solved in the plane in $O(\sqrt{n} \log n)$ time using $O(n)$ space. Matoušek [10] has shown how to achieve $O(n^{1-1/d})$ query time for simplex range searching with nearly linear space. This is close to Chazelle's lower bound of $\Omega(n^{1-1/d}/\log n)$ [6] for linear space. For halfspace range queries, Brönnimann, et al. [3] give a lower bound of $\Omega(n^{1-2/(d+1)})$ (ignoring logarithmic factors) assuming linear space. This lower bound applies to the more general case of spherical range queries as well.

Unfortunately, the lower bound arguments defeat any reasonable hope of achieving polylogarithmic performance for arbitrary (nonorthogonal) ranges. This suggests that it may be worthwhile considering variations of the problem, which may achieve these better running times. In this paper we consider an approximate version of range searching. Rather than approximating the count, we consider the range to be a *fuzzy* range, and that data points that are "close" to the boundary of the range (relative to the range's diameter) may or may not be included in the count.

To make this idea precise, we assume that ranges are bounded sets of bounded complexity. (Thus our results will not be applicable to halfspace range searching). Given a range Q of diameter s, and given $\epsilon > 0$, define Q^- to be the locus of points whose distance from a point exterior to Q is at least $s\epsilon$, and Q^+ to be the locus of points whose distance from a point interior to Q is at most $s\epsilon$. (Equivalently, Q^+ and Q^- can be defined in terms of the Minkowski sum of a ball of radius s and either Q or its complement.) Define a *legal answer* to an ϵ-approximate range query to be *weight*(P') for any subset P' such that

$$P \cap Q^- \subseteq P' \subseteq P \cap Q^+.$$

This definition allows for two-sided errors, by failing to count points that are barely inside the range, and counting points barely outside the query range. It is trivial to modify the algorithm so that it produces one-sided errors, forbidding either sins of omission or sins of commission (but obviously not both).

Approximate range searching is probably interesting only for fat ranges. Overmars [11] defines an object Q to be *k-fat* if for any point p in Q, and any ball B with p as center that does not fully contain Q in its interior, the portion of B covered by Q is at least $1/k$. For ranges that are not k-fat, the diameter of the range may be arbitrarily large compared to the thickness of the range at any point. However, there are many applications of range searching that involve fat ranges.

There are a number of reasons that this formulation of the problem is worth considering. It is well known that what seems to make range queries "hard" to solve are the points that are near the boundary of the range. However, there are many applications where data are imprecise, and ranges themselves are imprecise. For example, the user of a geographic information system that wants to know how many single family dwellings lie within a 60 mile radius of Manhattan, may be quite happy with an answer that is only accurate to within a few miles. Also range queries are often used as part of an initial filtering process to very large data sets, after which some more complex test will be applied to the points within the range. In these applications, a user may be quite happy to accept a coarse filter that runs faster. The user is free to adjust the value of ϵ to whatever precision is desired (without the need to apply preprocessing again), with the understanding that a tradeoff in running times is involved.

In this paper we show that by allowing approximate ranges, it is possible to achieve significant improvements in running times, both from a theoretical as well as practical perspective. We show that (for fixed dimension) after $O(n \log n)$ preprocessing, and with $O(n)$ space, ϵ-approximate range queries can be answered in time $O(\log n + 1/\epsilon^d)$. Under the assumption that ranges are convex, this can be strengthened to $O(\log n + 1/\epsilon^{d-1})$. Some of the features of our method are

- The data structure and preprocessing time are independent of the space of possible ranges and ϵ. We only assume that in constant time (depending on dimension) it is possible to determine whether there is a nonempty intersection between the inner and outer ranges (Q^- and Q^+) and a cube.

- Space and preprocessing time are free of exponential factors in dimension. Space is $O(dn)$ and preprocessing time is $O(dn \log n)$. (Assuming the binarized version of the data structure, discussed in Section 2.)

- The algorithms are quite simple. The data structure is a variant of the well-known quadtree data structure.

- Our experimental results show that even for uniformly distributed points in dimension 2, there is a significant improvement in the running time if a small approximation error is allowed. Furthermore, on average the *effective error* (defined in Section 5) committed by the algorithm is much smaller than the allowed error, ϵ.

We also present lower bound of $\Omega(\log n + 1/\epsilon^{d-1})$, for the complexity of answering ϵ-approximate range queries assuming a partition tree approach for cubical range in fixed dimension. Thus our approach is optimal under these assumptions for convex ranges.

2 The BBD Tree.

In this section we describe the data structure from which queries will be answered. We call this structure a *balanced box-decomposition tree* (or BBD tree) for the point set. The BBD tree is a balanced variant of a number of well-known data structures based on hierarchical subdivision of space into rectilinear regions. Examples of this class of structure include point quadtrees [13], *k-d* trees [2], or (unbalanced) box-decomposition tree (also called a fair-split tree) [1, 4, 8, 14].

Except for the *k-d* tree, none of these data structures need be balanced, in the sense that their depth is not bounded by $O(\log n)$. However, all of these data structures, except the *k-d* tree subdivide space into regions of bounded aspect ratio. The BBD tree achieves both of these properties. This is not the first example of such a tree. Arya et al. [1] showed that balance could be imposed on a box-decomposition by computing a centroid decomposition tree or topology tree for the box-decomposition tree. This idea was also used by Callahan and Kosaraju to modify the fair-split tree [5]. The BBD tree is a somewhat more direct implementation of this same idea. We present a brief description of this structure for the sake of completeness.

The BBD tree is a balanced 2^d-ary tree associated with a hierarchical subdivision of space into cells each of $O(d)$ complexity. For our purposes, a *box* in R^d is d-dimensional cube with faces orthogonal to the coordinate axes. We assume that all the data points have been scaled to lie within a d-dimensional unit hypercube H. This can be done in linear time before preprocessing. Define a *quadtree box* to be any box obtained by a finite number of applications of the following recursive rules.

- H is a quadtree box.
- If b is a quadtree box, then so are each of the 2^d boxes of half the side length formed by splitting b using d hyperplanes, each orthogonal to one of the coordinate axes.

Observe that each quadtree box has sides of length $(1/2)^i$ for some $i \geq 0$. All the boxes appearing in our BBD tree are quadtree boxes. Data points can be thought of as degenerate quadtree boxes of side length 0.

Each node v in the BBD tree is associated with a region denoted, *cell*(v). Each cell is the set theoretic difference of two quadtree boxes, a bounding box, *BB*(v), and a (possibly empty) inner box, *IB*(v), which is properly contained within *BB*(v). The set of data points associated with v, $P(v)$ is the intersection of P with v's cell, that is,

$$P(v) = P \cap (BB(v) - IB(v)).$$

The *size* of a node *size*(v) is the side length of its bounding box *BB*(v). Observe that the Euclidean distance between any two points in a cell of size s is at most $s\sqrt{d}$. The *weight* of a node, *weight*(v), is the cardinality of $P(v)$ (or more generally, the sums of weights of the points in the cell). We assume that each node v contains the quantities, *BB*(v), *IB*(v) (and a flag indicating whether *IB*(v) is empty), *size*(v), *weight*(v).

The root of the tree is associated with the bounding hypercube H, and its inner box is empty. Leaf nodes are each associated with a single data point (or generally a small constant number of data points), and have no inner box. Internal nodes are two types, *splitting nodes* and *shrinking nodes*, according to the way in which they subdivide the associated cell. Each splitting node v subdivides the associated cell by splitting its bounding quadtree box into 2^d quadtree boxes each of half the size by the process described earlier. These smaller boxes are associated with each of the 2^d children of v. By the nature of quadtree boxes, the inner box of v will either be entirely contained within one of v's children, or it will be equal to one of these boxes. In the former case, the inner box is made the inner box of the appropriate child cell, and all the other children have no inner box. In the latter case, the corresponding child is null (because the associated region is empty).

Each shrinking node v subdivides the associated cell by splitting it about a box b which contains the inner box of v (if any). The two children of v consist of one cell whose bounding box is $BB(v)$ and inner box b, and the other cell whose bounding box is b and inner box is $IB(v)$. (Note that shrinking is different from the shrinking operation defined in [1], because the region between the outer and inner box need not be free of data points.) These operations are illustrated in Figure 1(a) and (b).

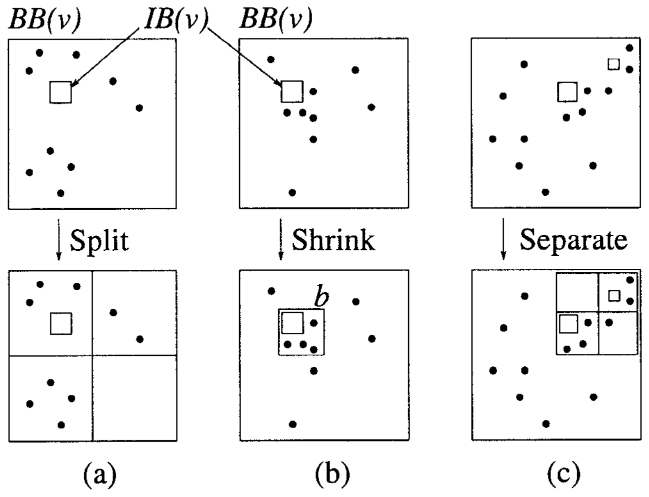

Figure 1: Splitting and shrinking.

For the purposes of implementation in higher dimensions, nodes with 2^d children are quite impractical. This can be overcome by implementing each split by a series of d consecutive binary splits, each cutting along a plane orthogonal to a different coordinate axis. The resulting boxes are not necessarily cubes, but they have bounded aspect ratio. However, for the purposes of describing our algorithms, it is somewhat simpler to think of splitting as an atomic operation, rather than this alternative *binarized* tree.

We claim that for every cell it is possible to apply a shrinking operation so that each of the resulting subcells contains at most a fraction of $2^d/(2^d+1)$ of the points in the original cell. To see this, consider a cell with m points, and consider a sequence of consecutive splits, starting with the original cell. After each split, apply the next split to the subcell with the largest number of points. Repeat until the largest subcell c has at most $m2^d/(2^d+1)$ points. It is easy to show that at most $m2^d/(2^d+1)$ points can lie outside c. Thus, it is always possible to find a shrinking cell containing a constant fraction of the points in a given cell. Such a shrinking operation is called a *centroid shrink.* (The term arises from the fact that the box c corresponds to finding a centroid node in the unbalanced box-decomposition tree, as described in [1].)

However, performing a centroid shrink may result in two inner boxes within the same bounding box. Repeated application could result in cells of arbitrary complexity. To remedy this, whenever a centroid shrink results in a box which is disjoint of the existing inner box, we separate these two inner boxes as follows. First, we find the smallest quadtree box which contains both of these boxes. We shrink to this box, and then we split this box. By the minimality of the containing box, the two inner boxes will become inner boxes of two different boxes in the split (or they will be equal to one of these boxes, in which case the box can be ignored as a child). This process of separating inner boxes increases the size and depth of the tree by a constant factor. However, it has the nice feature that each cell in the tree consists of a single bounding box and a (possibly empty) inner box, and so has complexity $O(d)$.

By alternating splitting rules, first applying a split followed by a centroid shrink (and box separation), and repeating until each cell contains at most a single point, we produce the balanced box-decomposition tree (BBD tree). The following lemma summarizes the main elements of the BBD tree.

Lemma 1 *Given a set of n points in a d-dimensional unit hypercube:*

(i) the BBD tree has height $O(d \log n)$,

(ii) each node is associated with a cell of complexity $O(d)$,

(iii) with every 2 levels of descent in the tree the size of the associated cells decreases by at least a factor of $1/2$.

The following lemma, which follows from the fact that cells cannot be too "skinny" will also be of importance. This type of *packing lemma* is common to many analyses of box-decomposition trees. The proof is omitted from this version.

Lemma 2 *The number of cells in a BBD tree of size at least s with pairwise disjoint interiors, and which intersect a range of diameter* $2r$ *is at most* $(1+\lceil\frac{2r}{s}\rceil)^d$.

Lemma 3 *The number of cells in a BBD tree of size at least s with pairwise disjoint interiors, and which intersect the boundary of a convex range of diameter* $2r$ *is at most* $O(1+\lceil\frac{2r}{s}\rceil)^{d-1}$.

Lemma 4 *The BBD tree can be built in* $O(n \log n)$ *time and* $O(n)$ *space.*

Proof: (Sketch) The construction of the BBD tree follows largely from the construction described by Callahan and Kosaraju for the fair-split tree [4]. The only nontrivial step not described there is the implementation of the centroid shrink. The problem is that it is not generally possible to bound the number of split operations needed until the number of points falls below some constant (for example, if the points are densely clustered). To do this, it suffices to determine the smallest quadtree box enclosing the current set of points at each step. This can be done in constant time, under the assumption that the model of computation supports the operations of exclusive-or, integer logarithm, and integer division on the coordinates of the data points [1]. Callahan and Kosaraju have shown that this assumption can be overcome with a somewhat more careful choice of splitting rules [5]. □

3 Range Searching Algorithm.

In this section we present the algorithm for answering range queries using the BBD tree. For simplicity, we consider the case where the query range is a ball of radius r centered at a query point q, but the generalization to other types of ranges is straightforward. We use the terms *inner range* and *outer range* to refer to the balls of radius $r^- = r/(1+\epsilon)$ and radius $r^+ = r(1+\epsilon)$ centered at q. Although we assume that $\epsilon > 0$ for the purposes of analysis, the algorithm runs correctly even if $\epsilon = 0$.

Generalizing range search algorithms for partition trees, the main idea of the algorithm is to simply descend the tree and classify nodes as lying completely inside the outer range or completely outside the inner range. If a node cannot be classified we recursively explore its children. The algorithm starts with the root of the tree. Let v denote the current node being visited.

(1) If *cell*(v) lies completely inside outer range, then return(*weight*(v)).

(2) If *cell*(v) lies completely outside inner range then return(0).

(3) If v is a leaf node, check for each associated point whether it lies inside the true range. Return(m), where m is the number of points that do lie inside.

(4) Otherwise, recursively call the procedure with the left and right child and return the sum of the two weights obtained.

The correctness of this simple algorithm is quite straightforward, and has been omitted from this version.

The main result of this section is the following theorem, which establishes the running time of the range counting algorithm.

Theorem 1 *After* $O(n \log n)$ *preprocessing time, and data structure of size* $O(n)$ *can be built, so that given a spherical query range and* $\epsilon > 0$, *a* $(1+\epsilon)$*-approximate range count can be computed in* $O((\log n) + 1/\epsilon^d)$ *time. (Constant factors in preprocessing time and space are linear in d, and constant factors in query time are on the order of* $d^2 2^d$*.)*

Proof: Preprocessing has already been discussed. We start with two definitions. A node v is said to be *visited* if the algorithm is called with node v as argument. A node v is said to be *expanded* if the the algorithm visits the children of node v. We distinguish between two kinds of expanded nodes depending on size. An expanded node v for which *size*(v) $\geq 2r$ is *large* and otherwise it is *small.* We will show that the number of large expanded nodes is bounded by $O(2^d \log n)$ and the number of small expanded nodes is bounded by $O((2\sqrt{d}/\epsilon)^d)$. Because each node can be expanded in $O(2^d)$ time, it will follow that the total running time of the algorithm is $O(2^d(2^d \log n + (2\sqrt{d}/\epsilon)^d))$. A factor of 2^d can be saved in expansion time if the 2^d-ary tree is replaced by a binarized tree described in Section 2, while adding a factor of d in the depth of the tree and a factor of d to the expansion time. This yields a better total time of $O(d^2 2^d \log n + d(2\sqrt{d}/\epsilon)^d)$. In either case, the running time is $O(\log n + 1/\epsilon^d)$ for fixed d.

We first show the bound on the number of large expanded nodes. In the descent through the BBD tree, the sizes of nodes decrease monotonically. Consider the set of all expanded nodes of size greater than $2r$. These nodes induce a subtree in the BBD tree. Let V denote the leaves of this tree. The cells associated with the elements of V are pairwise disjoint from one another, and furthermore they intersect the range (for otherwise they would not be expanded). It follows from Lemma 2 (applied to the cells associated with V) that there are at most $(1 + \lceil 2r/(2r) \rceil)^d = 2^d$, such boxes. Because the depth of the tree is $O(\log n)$, the total number of expanded large nodes is $O(2^d \log n)$, as desired.

Next we bound the number of small expanded nodes. First we claim that any node of size less than $r\epsilon/\sqrt{d}$ cannot be expanded. For a node to be expanded its cell must intersect the inner range of radius $r^- = r/(1+\epsilon)$ and the complement of the outer range of radius $r^+ = r(1+\epsilon)$. Hence the cell must have diameter of at least $r^+ - r^- \geq r\epsilon$. Since the diameter of a cell of size s is at most $s\sqrt{d}$, a cell of size less than $r\epsilon/\sqrt{d}$ is too small to be expanded.

To complete the analysis of the number of small expanded nodes, it suffices to count the number of expanded nodes of sizes from $2r$ down to $r\epsilon/\sqrt{d}$. Because sizes are powers of $1/2$, it suffices to count the number of expanded nodes of *size group* $(1/2)^i$, where i varies over an appropriate range. Consider the expanded nodes in the i-th size group. Because these nodes have the same size, the corresponding cells have pairwise disjoint interiors, and they overlap the query range. Applying Lemma 2, it follows that the number of maximal nodes in the i-th group is $(1 + \lceil 2^{i+1} r \rceil)^d$. Thus the total number of expanded nodes in all the size groups is at most

$$\sum_{i=a}^{b} \left(1 + \left\lceil 2^{i+1} r \right\rceil\right)^d,$$

where $a = -\lg 2r$ and $b = -\lg(r\epsilon/\sqrt{d})$. This is a geometric series, which is dominated asymptotically by its largest term,

$$\left(1 + \left\lceil \frac{2r\sqrt{d}}{r\epsilon} \right\rceil\right)^d = O\left(\left(\frac{2\sqrt{d}}{\epsilon}\right)^d\right),$$

as desired. □

For convex ranges, we can easily show a tighter bound of $O((\log n) + 1/\epsilon^{d-1})$ by using Lemma 3 in place of Lemma 2.

4 Lower Bounds

The method we use in this paper to solve the approximate range counting problem falls under the partition tree paradigm. This paradigm is also commonly used for solving the exact version of this problem. In the context of exact range counting, Chazelle and Welzl [7] have developed an interesting lower bound argument for any algorithm that uses partition trees. In this section we develop a similar argument for the approximate problem, which will establish the optimality of our algorithm in this paradigm.

We start by reviewing the notion of a partition tree. We are given a set P of n data points. A partition tree is a rooted tree of bounded degree in which each node v of the tree is associated with a set of points $P(v)$, according to the following rules. (For simplicity we will assume that the degree is at least two; it will be easy to see that the argument we develop here also holds without this assumption.)

(a) The leaves of the tree have a one-to-one correspondence with the data points.

(b) The set associated with an internal node v is formed by taking the union of all the points in the leaves of the subtree rooted at v.

With each node v we also store its *weight*(v) defined as the cardinality of the set $P(v)$. Given a range Q we can recursively search the partition tree to count the number of points inside Q as follows. We start at the root of the tree and initialize a global variable *count* to 0. At a node v we do the following.

(a) if $P(v) \subset Q$, we add *weight*(v) to the count.

(b) if $P(v) \cap Q = \emptyset$, we do nothing.

(c) Otherwise, we recursively search its children.

It is easy to see that the algorithm correctly solves the range counting query. Assuming that the conditions in Step (a) and (b) can be carried out in $O(1)$ time, the number of nodes visited by the algorithm accurately reflects its running time. Chazelle and Welzl [7] have shown that in the worst case the number of nodes visited is $\Omega(n^{1-1/d})$ for *any* partition tree. Using similar techniques, we show a lower bound on the number of nodes visited for the approximate version of the problem.

First we modify the above algorithm to solve the approximate range counting query. Let Q be the query range and let Q^+ and Q^- be the ϵ expansion and contraction of this range, respectively, as defined in the introduction. For the approximate range counting problem, we make a few straightforward modifications to the search algorithm given above.

(a) if $P(v) \subset Q^+$, we add *weight*(v) to the count.

(b) if $P(v) \cap Q^- = \emptyset$, we do nothing.

(c) Otherwise, we recursively search its children.

Define the *visiting number* of a partition tree as the maximum number of nodes visited by the above algorithm over all query ranges. We show a lower bound of $\log n + (1/\epsilon)^{d-1}$ on the visiting number of any partition tree. First, we modify some of the definitions of Chazelle and Welzl [7] to apply to the approximate problem. We say that a set $P(v)$ is *stabbed* if neither $P(v) \subset Q^+$ nor $P(v) \cap Q^- = \emptyset$ is true. In other words, $P(v)$ contains both a point inside Q^- and a point outside Q^+. We define the stabbing number of a spanning path as the maximum number of edges on the path (each edge is a set of its two end points) that are stabbed. Here the maximum is computed over all query ranges. Along the lines of Lemma 3.1 in [7], we can easily establish the following.

Lemma 5 *If T is any partition tree for P, then there exists a spanning path whose stabbing number does not exceed the visiting number of T.*

We will now exhibit a point set in d dimensions and a set of query ranges such that any spanning path will have a stabbing number of at least $1/\epsilon^{d-1}$ with respect to one of the query ranges. We assume that the dimension d is fixed. Consider a unit hypercube $[0,1]^d$ divided into a regular grid consisting of k^d cells of equal size. We choose $k = \lceil 1/(4\epsilon) \rceil$ and use $\epsilon' = 1/k$ to denote the grid spacing. The data set consists of one point located at each of the vertices of the grid, which gives a total of $\Omega(1/\epsilon^d)$ points. The query ranges in our set are balls in the L_∞ metric of radius unity. The centers of these balls are located along the d principal axis at distances from the origin of $-1+\epsilon'/2, -1+3\epsilon'/2, \ldots, -\epsilon'/2$, respectively. This gives a total of $O(d/\epsilon)$ query ranges. Now consider any spanning path on the set of points P. From the construction it is straightforward to show that every edge on this spanning path is stabbed by at least one query range. Thus the number of stabbed edges on the spanning path equals the total number of edges on the path which is $\Omega(1/\epsilon^d)$. Dividing this by the total number of query ranges implies that a query range stabs on average $\Omega(1/(d\epsilon^{d-1}))$ edges of the spanning path. Therefore there must exist a query range which stabs $\Omega(1/(d\epsilon^{d-1}))$ edges. Thus the stabbing number of any spanning path exceeds this quantity. Combining this with Lemma 5 gives us a lower bound on the visiting number of partition trees.

We next show a $\Omega(\log n)$ lower bound on the visiting number of any partition tree. Consider a set of n distinct data points and a set of n query ranges consisting of L_∞ ball centered at each of these points. The radius of these balls is chosen to be sufficiently small so that the $(1+\epsilon)$ expansion of the ball contains no other point. It is easy to see that the ball centered at p stabs the point sets corresponding to every proper ancestor of p. Since there must be a leaf at depth $\Omega(\log n)$, this gives a lower bound on the number of nodes whose point sets are stabbed by the ball centered at the point associated with the leaf. All such nodes are visited by the algorithm, hence this is also a bound on the visiting number of the partition tree. Combining this with the results of the last paragraph, we have the following lower bound on the visiting number of any partition tree. In fact, the lower bound holds even under the restriction to L_∞ balls.

Theorem 2 *For the set of query ranges consisting of balls in the L_∞ metric, the visiting number of any partition tree exceeds $\Omega(\log n + 1/\epsilon^{d-1})$.*

The theorem implies the optimality of our algorithm in the partition tree paradigm for convex ranges, and near optimality for the more general class of query ranges discussed in the introduction.

5 Experimental Results

To establish the validity of our claims empirically, we implemented our algorithm and tested it on a number of data sets of various sizes, various distributions, and with various sizes and types of ranges. To enhance performance, we implemented a variation of the data structure described in Section 2. First, we implemented the *binarized* version of the tree mentioned in this section (each node has two children rather than 2^d). Second, we did not always split boxes through the midpoint, but used a somewhat more sophisticated decomposition method, called the *fair-split rule* [1]. Intuitively, this splitting rule attempts to partition the point set of each box as evenly as possible, subject to maintaining boxes with bounded aspect ratio. Finally, our decomposition process attempted to *avoid* centroid shrinking whenever it was not warranted. The reason is that there are optimizations that can be performed at splitting nodes that are not possible at shrinking nodes. As long as the fair-split rule produced trees whose depth was within a constant factor of $\log_2 n$, we did not introduce centroid shrinking, and for the data sets we tested it was never neeed.

We ran our program for approximate range counting for ϵ ranging from 0 (exact searches) to 0.5. Our experiments were conducted for data points drawn from a number of distributions. Due to space limitations, only the following two are presented in this paper.

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

ClusNorm: Ten points were chosen from the uniform distribution over the unit hypercube and a Gaussian distribution with standard deviation 0.05 centered at each.

For each distribution we generated data sets ranging in size from $2^6 = 64$ to $2^{16} = 65,536$. Experiments were run in dimensions 2 and 3, and the query ranges were either L_2 balls (circles) or L_∞ balls (squares). Due to space limitations, we only show the results for dimension 2 and for circular ranges. We tested radii, ranging in size from $1/256$ to $1/2$. For each experiment, we fixed ϵ and the radius of the query balls and measured a number of statistics, averaged over 1,000 queries. The center of the query ball was chosen from the same distribution as the query points.

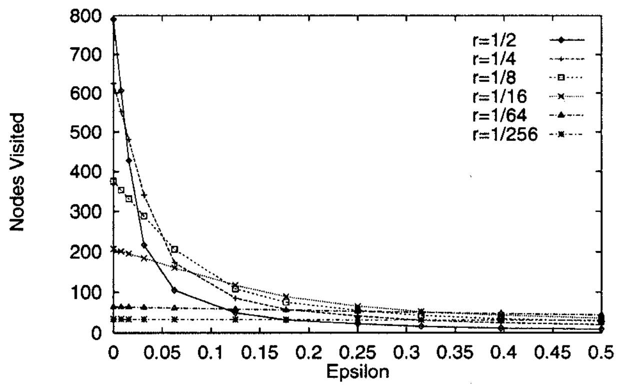

Figure 2: Number of nodes visited vs. ϵ. Uniform distribution.

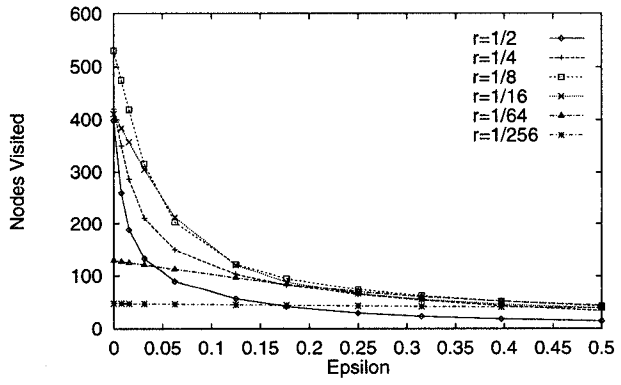

Figure 3: Number of nodes visited vs. ϵ. Clustered normal distribution.

In Figures 2 and 3, for each of the distributions, we show the number of nodes visited as a function of the accuracy of approximation, ϵ, for 65,536 data point. Since the algorithm does a constant amount of work for each node visited, the average number of nodes visited accurately reflects its running time. (We also measured floating point operations, and found that in dimension 2 on the average there were from 10 to 20 floating point operations for each node visited.) The key observation is that as ϵ increases (even to relatively small values in the range from 0.05 to 0.1), there are significant improvements in running time (factors as high as 10 to 1, and often around 4 to 1) for larger ranges. As ϵ grows, the running times tend to converge, irrespective of radius. Improvements for smaller ranges were not as significant, because the running times on small ranges are uniformly small. Results for square ranges were similar, and results in 3-space were similar, although the improvements were not quite as dramatic.

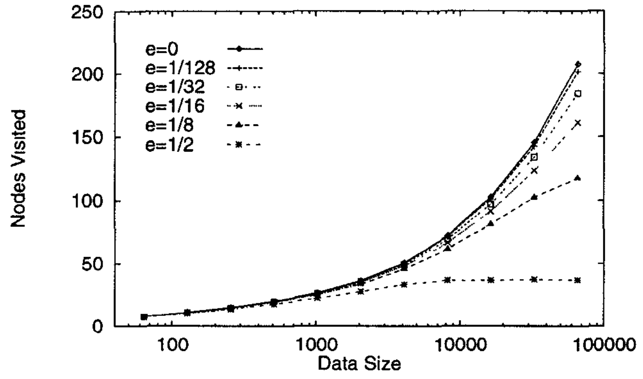

Figure 4: Number of nodes visited vs. number of data points. Uniform distribution.

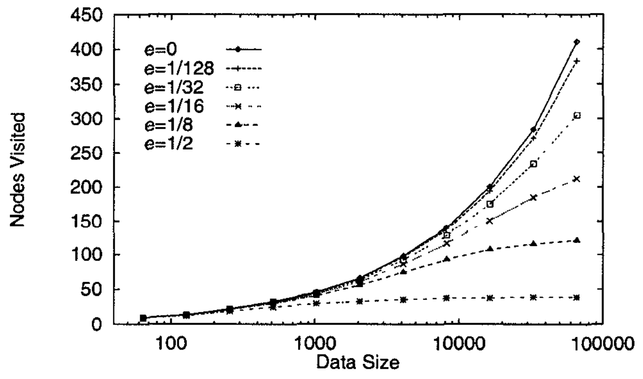

Figure 5: Number of nodes visited vs. number of data points. Clustered normal distribution.

In Figures 4 and 5 we show the number of nodes visited as a function of the number of data points. Note that the x-axis is given on a log scale. The key observation is that, as ϵ increases, the running times show a decreasing dependence on the number of data points. We also ran the experiments for smaller values of radius, and observed that the decrease of dependence occurs, but for larger values of ϵ or n. Results with square ranges and in 3-space showed a similar behavior.

We measured one interesting statistic, called *effective error* or *effective epsilon.* Consider a range of radius r and a point at distance r'. If $r' < r$ but the point was classified as being outside the range, the associated *misclassification error* is defined to be the relative error, $(r - r')/r'$; and if $r' > r$ but the point was classified as being inside the range, the associated *misclassification error* is $(r' - r)/r$. By definition, there can be no classification error greater than ϵ. But the algorithm may be doing better than this. To see how much better it is doing, we measured this relative error for every misclassified point, and averaged this over all the points which were eligible for misclassification (that is, points lying in the difference of the outer and inner ranges). This quantity is the *effective error* of the query. If no points were eligible for misclassification, then this quantity is zero.

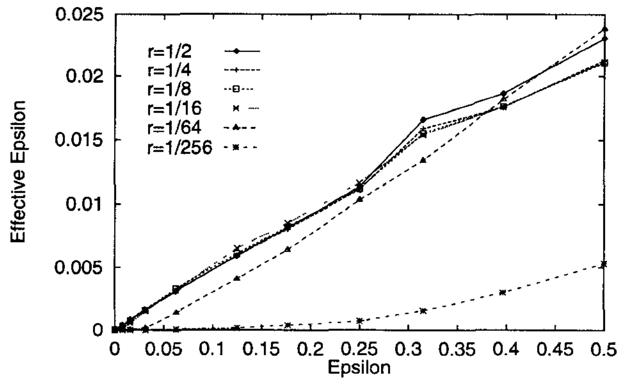

Figure 6: Effective error vs. ϵ. Uniform distribution.

In Figures 6 and 7 we show the effective errors as a function of ϵ, for 65,536 data points. The key observation is that effective error appears to vary almost linearly with ϵ (depending on distribution, dimension, and other factors). In dimension 2, effective errors were frequently less than 0.06ϵ, and in all distributions effective error was never greater

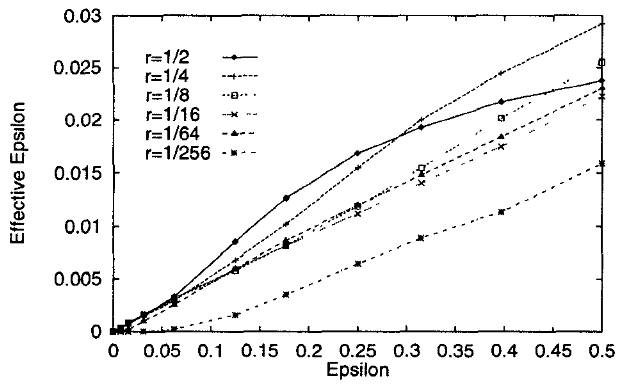

Figure 7: Effective error vs. ϵ. Clustered normal distribution.

than 0.1ϵ. These bounds were observed across all distributions tested, for both circular and square ranges, and in both dimensions 2 and 3. This explains in part, one of the reasons that we ran experiments with such large values of ϵ. Even with ϵ as large as 0.5 (allowing a maximum 50% error), we were often observing much smaller effective errors in the range of 1.5% to 3%.

References

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 573–582, 1994.

[2] J. L. Bentley. K-d trees for semidynamic point sets. In *Proc. 6th Ann. ACM Sympos. Comput. Geom.*, pages 187–197, 1990.

[3] H. Brönnimann, B. Chazelle, and J. Pach. How hard is halfspace range searching. *Discrete Comput. Geom.*, 10:143–155, 1993.

[4] 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 Annu. ACM Sympos. Theory Comput.*, pages 546–556, 1992.

[5] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, 1995.

[6] B. Chazelle. Lower bounds on the complexity of polytope range searching. *J. Amer. Math. Soc.*, 2:637–666, 1989.

[7] B. Chazelle and E. Welzl. Quasi-optimal range searching in spaces of finite VC-dimension. *Discrete Comput. Geom.*, 4:467–489, 1989.

[8] 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.

[9] N. Farvardin and J. W. Modestino. Rate-distortion performance of DPCM schemes for autoregressive sources. *IEEE Transactions on Information Theory*, 31:402–418, 1985.

[10] J. Matoušek. Range searching with efficient hierarchical cuttings. *Discrete Comput. Geom.*, 10(2):157–182, 1993.

[11] M. H. Overmars. Point location in fat subdivisions. *Inform. Process. Lett.*, 44:261–265, 1992.

[12] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, New York, NY, 1985.

[13] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison Wesley, Reading, MA, 1990.

[14] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete Comput. Geom.*, 4:101–115, 1989.