Approximate Range Searching[*]

Sunil Arya[†]
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

David M. Mount[‡]
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

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 w and $\epsilon > 0$, an approximate range query treats the range as a fuzzy object, meaning that points lying within distance ϵw of the boundary of Q either may or may not be counted. We show that in any fixed dimension d, a set of n points in $\mathbf{R}^d$ 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 for approximate range searching based on partition trees of $\Omega(\log n + (1/\epsilon)^{d-1})$, which implies optimality for convex ranges (assuming fixed dimensions). Finally we give empirical evidence showing that allowing small relative errors can significantly improve query execution times.

[*]A preliminary version of this paper appeared in the *Proc. of the 11th Annual ACM Symp. on Computational Geometry*, 1995, 172–181.

[†]Email: `arya@cs.ust.hk`. The work of this author was partially supported by HK RGC grant HKUST 736/96E. Part of this research was conducted while the author was visiting the Max-Planck-Institut für Informatik, Saarbrücken, Germany.

[‡]Email: `mount@cs.umd.edu`. This author was supported by the National Science Foundation under grant CCR–9712379 .

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, $\mathbf{R}^d$, 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 $P \cap Q$, $\mathit{weight}(P \cap Q)$, 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., [15]). Chazelle and Welzl [9] showed that triangular range queries can be solved in the plane in $O(\sqrt{n} \log n)$ time using $O(n)$ space. Matoušek [13] 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 $\Omega(n^{1-1/d}/\log n)$ [8] for linear space. For halfspace range queries, Brönnimann et al. [5] gave 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 assume 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 w, and given $\epsilon > 0$, define the *inner range* Q^- to be the locus of points whose Euclidean distance from a point exterior to Q is at least $w\epsilon$, and define the *outer range* Q^+ to be the locus of points whose distance from a point interior to Q is at most $w\epsilon$. (Equivalently, Q^+ and Q^- can be defined in terms of the Minkowski sum and difference, respectively, of a ball of radius $w\epsilon$ with Q.) Define a *legal answer* to an ϵ-approximate range query to be $\mathit{weight}(P')$ for any subset P' such that

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

(See Fig. 1).

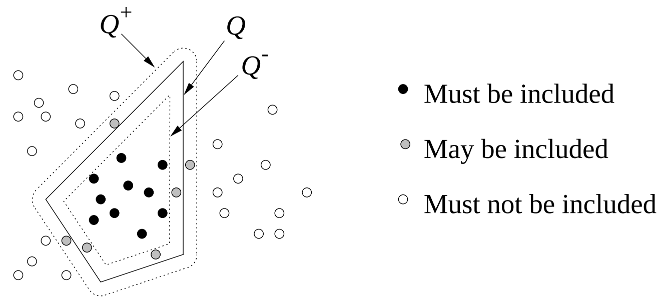

Figure 1: Approximate range searching queries.

This definition allows for two-sided errors, by failing to count points that are barely inside the range, and counting points barely outside the range. It is trivial to modify the algorithm so that it produces one-sided errors (thus forbidding either sins of omission or sins of commission, but obviously not both). Our results can be generalized to other definitions Q^- and Q^+ provided that the minimum boundary separation distance with Q is at least $w\epsilon$. We assume that the following *range-testing primitives* are provided:

Point membership in Q: whether a point p lies within Q,

Box intersection with Q^-: whether an axis-aligned rectangle has a non-empty intersection with Q^-, and

Box containment in Q^+: whether an axis-aligned rectangle is contained within Q^+.

Our running times are given assuming each of these can be computed in constant time. (In general it is the product of the maximum time for each primitive and our time bounds.) Our algorithm can easily be generalized to report the set of points lying within the range, and the running time increases to include the time to output the points.

Approximate range searching is probably of most interest for fat ranges. Overmars [14] 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. 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 d, 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, we show that this can be strengthened to $O(\log n + 1/\epsilon^{d-1})$. (These expressions are asymptotic in n and $1/\epsilon$ and assume d is fixed. See Theorems 1 and 2 for the exact dependencies on dimension.) Some of the features of our method are:

- The data structure and preprocessing time are independent of the space of possible ranges and ϵ. (Assuming that the above range-testing primitives are provided at query time.)
- Space and preprocessing time are free of exponential factors in dimension. Space is $O(dn)$ and preprocessing time is $O(dn \log n)$.
- The algorithms are relatively simple and have been implemented. 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, the *average error* (defined in Section 5) committed by the algorithm is typically much smaller than the allowed error, ϵ.

We also present a 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 hypercube range queries in fixed dimension. Thus our approach is optimal under these assumptions for convex ranges (up to constant factors depending on d and ϵ).

2 The Balanced Box-Decomposition Tree.

In this section we review the data structure from which queries will be answered. The structure is a *balanced box-decomposition tree* (or *BBD-tree*) for the point set. The BBD-tree [2] 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 structures include point quadtrees

[16], and variants [4], *k-d* trees [3], or (unbalanced) box-decomposition tree (also called a fair-split tree) [6, 10, 18]. A related structure is the BAR tree [11], which partitions space into regions of bounded aspect ratio but may not be rectilinear. For completeness we review the BBD-tree, emphasizing the elements that are important for this application.

We begin with a few definitions. By a *rectangle* in $\mathbf{R}^d$ we mean the d-fold product of closed intervals on the coordinate axes. The *size* of a rectangle is the length of its longest side. We define a *box* to be a rectangle such that the ratio of its longest to shortest side, called its *aspect ratio*, is bounded by some constant, which for concreteness we will take to be 2.

Each node of the BBD-tree is associated with a region of space called a cell. In particular, define a *cell* to be either a box or the set theoretic difference of two boxes, one enclosed within the other. Thus each cell is defined by an *outer box* and an optional *inner box*. Each cell is associated with the set of data points lying within the cell. Cells are considered to be closed, and hence points which lie on the boundary between cells may be assigned to either cell. The *size* of a cell is the size of its outer box.

Inner boxes satisfy a property called *stickiness* [2]. Intuitively stickiness means that an inner box cannot be too close (relative to its own size) to any face of the outer box, unless it touches this outer face. More formally, an inner box is *sticky* if when it is surrounded in a grid-like fashion with $3^d - 1$ identical boxes, the interior of each of these boxes either lies entirely inside or outside the outer box. This property is needed for technical reasons in proving the packing constraint (Lemma 2 below). The main properties of the BBD-tree are summarized in the following lemma.

Lemma 1 *Given a set of n data points P in $\mathbf{R}^d$ and a bounding hypercube C for the points, in $O(dn \log n)$ time it is possible to construct a binary tree representing a hierarchical decomposition of C into cells of complexity $O(d)$ such that*

(i) The tree has $O(n)$ nodes and depth $O(\log n)$.

(ii) The cells have bounded aspect ratio, and with every $2d$ levels of descent in the tree, the sizes of the associated cells decrease by at least a factor of $1/2$.

(iii) Inner boxes are sticky relative to their outer boxes.

The rest of this section presents the highlights of the BBD-tree and its construction. The details of the proof are provided in [2]. (We present only the simpler midpoint split form of the tree here.) The tree is constructed through the recursive application of two partitioning operations, *splits* and *shrinks*. They represent two different ways of subdividing a cell into two smaller *child cells*. A split partitions a cell by an axis-orthogonal hyperplane.

A shrink partitions a cell by a box that lies within the original cell. It partitions a cell into two children, one lying inside this box and one lying outside. If a cell contains an inner box, then a split will never intersect the interior of this box. If a shrink is performed on a cell that has an inner box, then this inner box will lie entirely inside the partitioning box. The resulting cells clearly have complexity $O(d)$.

The BBD-tree is constructed through a combination of split and shrink operations. Recall that C is a hypercube that contains all the points of P. The root of the BBD-tree is a node whose associated cell is C and whose associated set is the entire set P. The recursive construction algorithm is given a cell and a subset of data points associated with this cell. Each stage of the algorithm determines how to subdivide the current cell, either through splitting or shrinking, and then partitions the points among the child nodes. This is repeated until the cell has at most one point (or more practically, a small constant number of points, called the *bucket size*).

Given a node with more than one data point, we first consider the question of whether we should apply splitting or shrinking. A simple strategy (which we will assume in proving our results) is that splits and shrinks are applied alternately. This will imply that both the geometric size and the number of points associated with each node will decrease exponentially as we descend a constant number of levels in the tree. A more practical approach, which we have used in our implementation, is to perform splits exclusively, as long as the cardinalities of the associated data sets decrease by a constant factor after a constant number of splits. If this condition is violated, then a shrink is performed instead. Our experience has shown that shrinking is only occasionally invoked, but it is important to guarantee efficiency with highly clustered point distributions.

Once it has been determined whether to perform a split or a shrink, the splitting plane or partitioning box is computed, by a method to be described later. We store this information in the current node, create and link the two children nodes into the tree, and then partition the associated data points between these children. Data points lying on the splitting boundary may be assigned to either child. This can be done in $O(dn)$ time by a procedure due to Vaidya [18] (see also [2]).

Splitting is performed by bisecting the box by a hyperplane that is orthogonal to its longest side. Ties may be broken arbitrarily. The resulting cells are called *midpoint boxes.* (See Fig. 2(a).) This is just a binary variant of the well-known quadtree/octree splitting rule, which splits a hypercube cell into 2^d identical hypercubes of half the original size [16]. This binary version is significantly more practical in higher dimensional spaces. It is easy to see that the resulting boxes will have aspect ratios of either 1 or 2. Stickiness is also easy to verify, since if an inner box has width w along some coordinate axis and does not intersect a face of the enclosing outer box, then it is at distance at least w from this face [2].

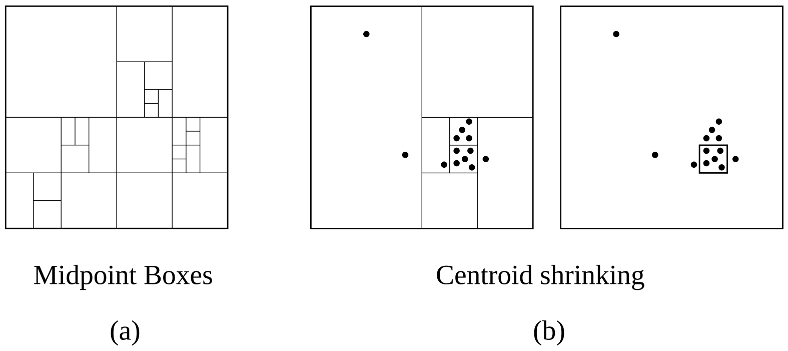

Figure 2: Midpoint cells and centroid shrinking.

Shrinking is performed as part of a global operation called a *centroid shrink*, which will generally produce up to three new nodes in the tree (two shrinking nodes and one splitting node). Let n_c denote the number of data points associated with the current cell. The goal of a centroid shrink is to decompose the current cell into a constant number of subcells, each containing at most $2n_c/3$ data points.

We begin with a simplified explanation of how centroid shrinking is performed, assuming the cell has no inner box. Without altering the current tree, repeatedly apply midpoint splits, always recursing on the child having the greater number of points. Repeat this until the number of points in the cell is no more than $2n_c/3$. The outer box of this cell is the partitioning box for the shrink operation. (The intermediate splits are discarded.) Observe that prior to the last split we had a box with at least $2n_c/3$ data points, and hence the partitioning box contains at least $n_c/3$ points. Thus, there are at most $2n_c/3$ points either inside or outside the partitioning box. (See Fig. 2(b).)

This procedure is not efficient as described. If the points are clustered in a very small region, then the number of midpoint splits needed until this procedure terminates cannot generally be bounded by a function of n_c. To remedy this problem before each split, compute the smallest midpoint box that contains the data points. (See Clarkson [10] for a solution to this problem based on floor, logarithm, and bitwise exclusive-or operations.) The very next split will succeed in producing a nontrivial partition of the points.

Now, suppose that the original cell had an inner box. We replace the single stage shrink described above with a 3-stage decomposition, which shrinks, then splits, then shrinks. Let b_I denote this inner box. When we compute the minimum enclosing midpoint box for the data points, we make sure that it includes b_I as well. Now we apply the above iterated shrinking/splitting combination, until (if ever) we first encounter a split that separates b_I from the box containing the majority of the remaining points. Let b denote the box that was just split. (See Figure 3(b).) We create a shrinking node whose partitioning box is b. We first shrink to node b, then apply this split, thus separating the majority points from b_I. Now that the inner box is eliminated, we simply proceed with the above procedure. If no split separates the b_I from the majority, then b_I will be nested within the partitioning box, which is fine.

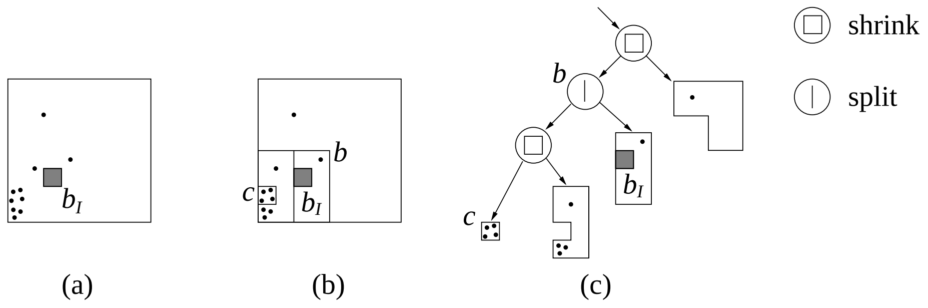

Figure 3: Midpoint construction: Centroid shrinking with an inner box.

We refer the reader to [2] for details of the $O(dn \log n)$ running time of the construction. The fact that we perform centroid splits every other level of the tree implies that the total number of nodes in the tree is $O(n)$, and the tree has $O(\log n)$ depth. It is easy to verify that the construction algorithm never produces two consecutive shrinks (since the 3-stage process interleaves a split between its shrinks). After any d splits the size of each node decreases by a factor of 1/2. Thus after descending $2d$ levels in the tree, cell sizes decrease by at least 1/2. Finally, by a simple postorder traversal, it is possible to label each node v in the tree with the weight of the associated points, denoted *weight*(v). This will be used in the query processing algorithm.

Before ending this section, we present some lemmas that form the basis for our later analysis. The first one is a key property of the BBD-tree, and follows from the fact that its cells are fat and inner boxes are sticky. It was proved in [2] for Minkowski balls, but we need a slightly different formulation for our purposes.

Lemma 2 (Packing Constraint) *Consider any set C of cells of the BBD-tree with pairwise disjoint interiors, each of size at least s, that intersect a range of diameter w. The size of such a set is at most $\left(1 + \left\lceil \frac{2w}{s} \right\rceil\right)^d$.*

Proof: From the 2:1 aspect ratio bound, the smallest side length of a box of size s is at least $s/2$. We first show that the number of disjoint boxes of side length at least $s/2$ that can overlap the range is at most $(1 + \lceil 2w/s \rceil)^d$. To see this, first scale space by a factor of $2/s$, implying that the range now has diameter $w' = 2w/s$ and each box has size at least 1. Consider a subdivision of $\mathbf{R}^d$ into an infinite integer grid of unit hypercubes. Observe that any range of diameter w' can be enclosed within a closed hypercube H whose vertices coincide with the integer grid and whose side length is at most $1 + \lceil w' \rceil$. (To see this, observe that the projection of the range onto any coordinate axis is an interval of length at most w', which can be contained within an integer range of width $1 + \lceil w' \rceil$.) There are $(1 + \lceil w' \rceil)^d = (1 + \lceil 2w/s \rceil)^d$ cubes of the grid enclosed within H. This is the desired upper bound, since no set of axis-aligned disjoint boxes of side length at least 1 can be packed more densely.

The above argument cannot be applied directly when shrinking cells are involved, because the outer box of a shrinking cell is not disjoint from its inner box. To complete the proof, we modify the set C, replacing each shrinking cell in the set with a box of size at least s, such that the resulting boxes are disjoint. Then the above argument is applied to complete the proof.

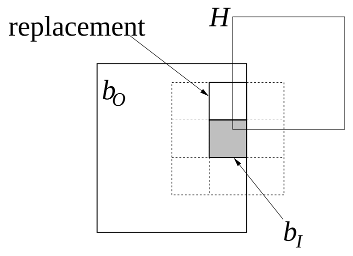

Figure 4: Packing constraint.

For each shrinking cell in C, if its inner box b_I is not in C, then we may replace this cell with its outer box, b_O. If b_I is in C, then consider the $3^d - 1$ identical boxes surrounding this box in a grid-like manner. (See Fig. 4.) All of these boxes are of size at least s. By the stickiness property, the interior of each of these boxes either lies entirely inside or outside b_O. Since H intersects both boxes, it must (by convexity) intersect at least one of the surrounding $3^d - 1$ boxes that lies within b_O. Replace b_O with this box in C. The replacement is of size at least s, it intersects H, and it is disjoint from b_I. After applying this to all shrinking cells, the resulting set of boxes are disjoint and satisfy the requirements of the lemma. Applying the above argument proves the result. □

In most applications of range searching, the range is a convex set. Next we show that if we add this restriction, the exponential dependence on d decreases slightly. The relevant quantity for our later analysis will be the number of cells that intersect the boundary of the range.

Lemma 3 *Consider any set C of cells of the BBD-tree with pairwise disjoint interiors, each of size at least s, that intersect the boundary of a convex range of diameter w. The size of such a set is at most* $2d^2 \left(1 + \left\lceil \frac{2w}{s} \right\rceil\right)^{d-1}$.

Proof: Let Q denote the range, and let ∂Q denote its boundary. Following the same reasoning as in the proof of Lemma 2, it suffices to bound the number of cells of an infinite square d-dimensional grid of side length $s/2$ that intersect ∂Q (and then apply the same replacement argument to generalize this to shrinking cells). As before we scale by $2/s$ so that the diameter of Q is $w' = 2w/s$, and enclose Q in an axis-aligned hypercube H with integer coordinates, whose width is $W = 1 + \lceil w' \rceil$. It suffices to show that ∂Q intersects at most $2d^2 W^{d-1}$ grid cells within H. Our proof is based on covering ∂Q with at most $2dW^{d-1}$ *covering sets*, such that each covering set intersects at most d grid cubes.

H has $2d$ faces each of dimension $(d-1)$. Let $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_{2d}$ denote the outward pointing normal unit vectors for each of these faces. Each of these vectors has exactly one nonzero coordinate, which is either 1 or -1. The grid naturally subdivides each of the faces of H into exactly W^{d-1} *subfaces*, where each is a $(d-1)$-dimensional unit hypercube, for a total of $2dW^{d-1}$ subfaces.

Because Q is convex, each point $p \in \partial Q$ can be associated with a supporting hyperplane passing through p. (If there are many then take any one.) Let $\mathbf{v}_p$ denote the outward pointing unit normal vector for this hyperplane. Given any other point on $q \in \partial Q$, the angle between vectors $\mathbf{v}_p$ and $q - p$ (the vector directed from p to q) is at least $\pi/2$. Let $\angle \mathbf{uv}$ denote the angle (in the range 0 to π) between vectors $\mathbf{u}$ and $\mathbf{v}$. We will make use of the fact that the angle between two nonzero unit vectors (i.e., the geodesic distance on a unit $d-1$ sphere) defines a metric on unit vectors. Hence the triangle inequality holds: $\angle \mathbf{wu} + \angle \mathbf{uv} \geq \angle \mathbf{wv}$.

The construction of the covering sets on ∂Q arises from the orthogonal projection of each subface onto Q in a direction parallel to the face's normal. To prevent each patch from intersecting many grid cubes, we first partition ∂Q into $2d$ regions, $R_1, R_2, \ldots, R_{2d}$, where each region is associated with a face of H. A point p is assigned to region R_i for which the angle $\angle \mathbf{v}_p \mathbf{u}_i$ is minimum. Ties may be broken arbitrarily. These regions are illustrated in Fig. 5(a). (Note that these regions may not be connected in general.)

First we claim that if p is assigned to region R_i, then the angle $\angle \mathbf{v}_p \mathbf{u}_i$ is at most $\arccos 1/\sqrt{d}$. That is, the angular distance from any unit vector to its nearest vector u_i cannot exceed this angle. To see this, first observe that by symmetry we may consider only vectors with positive coordinates. The closest coordinate vector (in angle) to any unit vector $\mathbf{v}$ corresponds to the largest coordinate of $\mathbf{v}$. To maximize the angle to the nearest coordinate vector, we should minimize its maximum coordinate. The unit vector achieving the maximum has all coordinates equal to $1/\sqrt{d}$. This vector forms the angle $\arccos 1/\sqrt{d}$ with its closest coordinate vector.

Consider the ith face F_i of H and any subface f lying on this face. Consider the set of points of R_i whose orthogonal projection onto F_i lies within f. The *covering set* associated with f is defined to be this set of

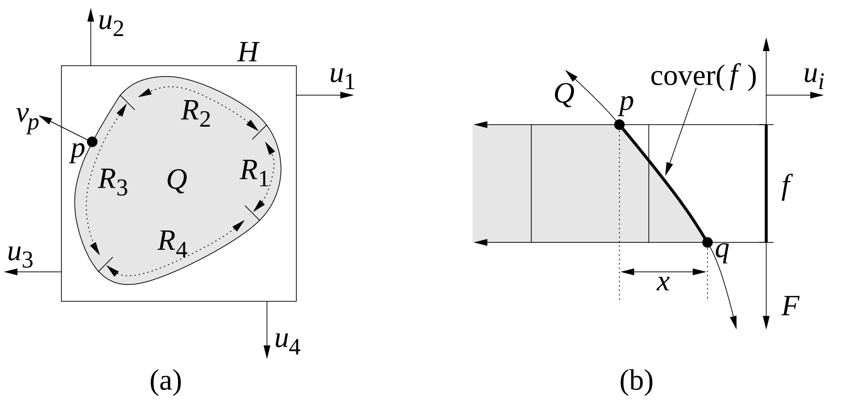

Figure 5: Proof of packing lemma for convex ranges.

points. Thus, ∂Q is covered by these $2dW^{d-1}$ sets.

All that remains to be shown is that each covering set overlaps at most d cubes of the grid. Consider the covering set for subface f and face F_i of H. The set lies within an infinite cylinder whose orthogonal cross section is f. (This is the shaded region in Fig. 5(b).) Let q and p be the closest and furthest points, respectively, of the covering set from f (or more formally, the limit points achieving these distances). Let x denote the difference in distances of q and p from f. We claim that it suffices to show that $x \leq d-1$, since if so, R_i cannot intersect more than $1 + \lceil x \rceil = d$ grid cubes within the cylinder.

To show that $x \leq d-1$, let $\mathbf{z}$ denote the vector from p to q, normalized to unit length. From the facts that the orthogonal projection of pq onto F lies within the subface f of diameter $\sqrt{d-1}$, and the orthogonal projection of pq onto $\mathbf{u}_i$ has length x, we have

$$\angle \mathbf{u}_i \mathbf{z} \leq \arctan \frac{\sqrt{d-1}}{x}.$$

Because p and q are both boundary points, from the observation above it follows that the angle $\angle \mathbf{v}_p \mathbf{z} \geq \pi/2$. Since p is in region R_i, we have $\angle \mathbf{v}_p \mathbf{u}_i \leq \arccos 1/\sqrt{d}$. Using this and the triangle inequality for angles we have

$$\arccos \frac{1}{\sqrt{d}} \;\geq\; \angle \mathbf{v}_p \mathbf{u}_i \;\geq\; \angle \mathbf{v}_p \mathbf{z} - \angle \mathbf{u}_i \mathbf{z} \;\geq\; \frac{\pi}{2} - \arctan \frac{\sqrt{d-1}}{x}.$$

Using the fact that $\arccos 1/\sqrt{d} = \arctan \sqrt{d-1}$, and some straightforward manipulations, we have $x \leq d-1$ as desired.

$\square$

3 Range Searching Algorithm.

In this section we present the algorithm for answering range queries using the BBD-tree. Let Q denote the range and w its diameter. Recall from the introduction, that the *inner range* Q^- and *outer range* Q^+ are the erosion and dilation, respectively, of Q by distance $w\epsilon$. Although we assume that $\epsilon > 0$ for the purposes of analysis, the algorithm runs correctly even if $\epsilon = 0$.

We generalize the standard 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, because it overlaps both ranges, then we recursively explore its children. The initial call to the recursive procedure is at the root of the BBD-tree. The procedure is shown in the Fig. 6.

Algorithm 1 [Query Processing]

Arguments: *A range Q, given with an inner range Q_I and an outer range Q_O, and a node v in the BBD-tree.*

Returns: *The total weight of points in the approximate range.*

```
function Query($Q, v$):
    if $cell(v) \subseteq Q_O$ then return $weight(v)$;
    if $cell(v) \cap Q_I = \emptyset$ then return $0$;
    if $v$ is a leaf then
        $w = 0$;
        for each $p \in points(v)$ do
            if $p \in Q$ then $w$ += $weight(p)$;
        return $w$;
    else return Query($Q, left(v)$) + Query($Q, right(v)$);
```

Figure 6: Query Processing Algorithm.

Observe that the algorithm can easily be modified to report the set of points (rather than their weight). The correctness of this algorithm follows immediately from the following lemma, and the observation that no point is counted twice because of the disjointness of the subtrees counted.

Lemma 4 *The query processing algorithm returns a count which includes all the points lying inside the inner range and excludes all the points lying* *outside the outer range.*

Proof: To establish the claim, let T denote the set of nodes visited for which the algorithm does not make a recursive call. The disjoint union of the cells corresponding to the set of nodes in T covers the hypercube C containing all the data points. Let p be some point lying inside the inner range Q^- and let v be the node of T which contains it. Then if v is a leaf node, p would be included in the count since it lies within Q. Otherwise if v is a non-leaf node, then, since node v does not result in a recursive call, *cell*(v) must lie completely inside Q^+. The point p would therefore have been included in the count returned in the algorithm's first step. In an analogous way, we can also show that the points outside the outer range are not included in the count, which proves the claim. □

The main result of this section is the following theorem, which establishes the running time of the range counting algorithm. Here we assume that the range-testing primitives can be executed in constant time.

Theorem 1 *Given a BBD-tree for a set of n data points in $\mathbf{R}^d$, given a query range Q of diameter w, and $\epsilon > 0$, ϵ-approximate range counting queries can be answered in $O(2^d \log n + d(3\sqrt{d}/\epsilon)^d)$ time. For fixed d this is $O(\log n + (1/\epsilon)^d)$.*

Proof: We start with some definitions. A node v is said to be *visited* if the algorithm is called with node v as an argument. A node v is said to be *expanded* if the algorithm visits the children of node v.

It will simplify matters to assume that everything has been scaled so that the enclosing hypercube C for the point set is a unit hypercube, implying that the cells of the BBD-tree have sizes that are positive integer powers of $1/2$. We distinguish between two kinds of expanded nodes depending on size. An expanded node v for which *size*$(v) \geq 2w$ 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 $O(d(3\sqrt{d}/\epsilon)^d)$. Since each node can be expanded and its children visited in constant time, it follows that the total running time is the sum of these two quantities.

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 $2w$. These nodes induce a subtree in the BBD-tree. Let L denote the leaves of this tree. The cells associated with the elements of L have pairwise disjoint interiors and they intersect the range (for otherwise they would not be expanded). It follows from Lemma 2 (applied to the cells associated with L) that there are at most $(1 + \lceil 2w/2w \rceil)^d = 2^d$, such boxes. By Lemma 1 the depth of the tree is $O(\log n)$, and hence 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 $2w\epsilon/\sqrt{d}$ cannot be expanded. For a node to be expanded its cell must intersect both the inner range Q^- and the complement of the outer range Q^+. Since the boundaries of Q^- and Q^+ are each separated from the boundary of Q by a distance of $w\epsilon$, they are separated from each other by a distance of $2w\epsilon$. Since the diameter of a cell of size s is at most $s\sqrt{d}$, a cell of size less than $2w\epsilon/\sqrt{d}$ cannot intersect both range boundaries and hence cannot be expanded.

Thus, it suffices to count the number of expanded nodes of sizes from $2w$ down to $2w\epsilon/\sqrt{d}$. To do this we group nodes in groups according to their size. For $i \geq 0$, define *size group* i to be the set of nodes whose cell size is $1/2^i$. Since small nodes are of size less than $2w$, the first size group of interest is $a+1$, where $1/2^{a+1} < 2w \leq 1/2^a$, and hence $a = \lfloor -\lg 2w \rfloor$. Since nodes that are smaller than $2w\epsilon/\sqrt{d}$ are not expanded, the last size group of interest is b, where $1/2^{b+1} < 2w\epsilon/\sqrt{d} \leq 1/2^b$, and hence $b = \left\lfloor -\lg(2w\epsilon/\sqrt{d}) \right\rfloor$. Because a node and its child may have the same size, we cannot apply the packing lemma directly to each size group. Define the *base group* for the ith size group to be the subset of nodes in the size group that are leaves or whose children are both in the next smaller size group. The cells corresponding to the nodes in a base group have pairwise disjoint interiors, since none of their descendents can be in the same base group. Applying Lemma 2, it follows that the number of nodes in the ith base group is at most

$$\left(1 + \left\lceil \frac{2w}{1/2^i} \right\rceil\right)^d = \left(1 + \left\lceil w2^{i+1} \right\rceil\right)^d.$$

From claim (ii) of Lemma 1 we know that at most $2d$ levels of ancestors above the base group can be in the same size group, and thus the number of nodes in any size group is at most $2d$ times the above quantity.

Thus, the total number of expanded nodes over all of the base groups is

$$E_d(w,\epsilon) \leq \sum_{i=a+1}^{b} \left(1 + \left\lceil w2^{i+1} \right\rceil\right)^d.$$

Observe that for $i \geq a+1$, we have $w2^{i+1} \geq w2^{a+2} \geq 1$. For any $x \geq 1$, note that $1 + \lceil x \rceil \leq 3x$, and hence we have

$$E_d(w,\epsilon) \;\leq\; \sum_{i=a+1}^{b} \left(3w2^{i+1}\right)^d \;\leq\; (6w)^d \sum_{i=0}^{b} (2^d)^i.$$

Solving this geometric series yields

$$\begin{aligned} E_d(w,\epsilon) \;&\leq\; (6w)^d \frac{(2^d)^{b+1} - 1}{2^d - 1} \;\leq\; (6w)^d \frac{2^d(\sqrt{d}/(2w\epsilon))^d}{2^d - 1} \\ &=\; \frac{2^d}{2^d - 1} \left(\frac{3\sqrt{d}}{\epsilon}\right)^d \;\leq\; 2\left(\frac{3\sqrt{d}}{\epsilon}\right)^d. \end{aligned}$$

This completes the proof. □

As mentioned earlier, range reporting queries can be answered in the same time, plus $O(m)$, where m is the number of points reported. Observe that the above proof is based on counting expanded nodes, which intersect both the inner and outer range, and hence intersect the boundary of Q. For convex ranges, we can use the same proof, but apply Lemma 3 to reduce the exponential dependence on dimension slightly.

Theorem 2 *Given a BBD-tree for a set of n data points in $\mathbf{R}^d$, given a convex query range Q of diameter w, and $\epsilon > 0$, ϵ-approximate range counting queries can be answered in $O(2^d \log n + d^3(3\sqrt{d}/\epsilon)^{d-1})$ time. For fixed d this is $O(\log n + (1/\epsilon)^{d-1})$.*

4 Lower Bounds for Approximate Range Searching

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 [9] 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 [17, 9]. 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.

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

(b) The subset of points 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.

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. 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 search the partition tree to count the number of points inside Q by a simple recursive procedure. Chazelle and Welzl [9] 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. We use the same natural generalization of the recursive range search procedure that was presented in Fig. 6.

Define the *visiting number* of a partition tree to be the maximum number of nodes visited by the above algorithm over all query ranges. We show that a lower bound of $\Omega(\log n + (1/\epsilon)^{d-1})$ holds in the worst case on the visiting number of any partition tree. First we modify some of the definitions of Chazelle and Welzl [9] to apply to the approximate problem. We say that a set $P(v)$ is *stabbed* if neither $P(v) \subseteq Q^+$ 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 can be stabbed by some query range. Along the lines of Lemma 3.1 in [9], 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 now exhibit a data set P_1 in d dimensions and a set of query ranges X such that any spanning path will have a stabbing number of at least $\Omega(1/\epsilon^{d-1})$ with respect to some query range in X. We assume that the dimension d is fixed. Consider an axis-aligned infinite grid with grid spacing 4ϵ. Assume that the origin is a vertex of the grid. The set P_1 consists of a data point at each vertex of the grid that lies within the unit hypercube $[0,1]^d$. Clearly, the number of data points is at least $\Omega(1/\epsilon^d)$. Let $\epsilon' = 4\epsilon$. The query ranges in set X are balls in the L_∞ metric of unit radius. 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 $O(1/\epsilon)$ query ranges.

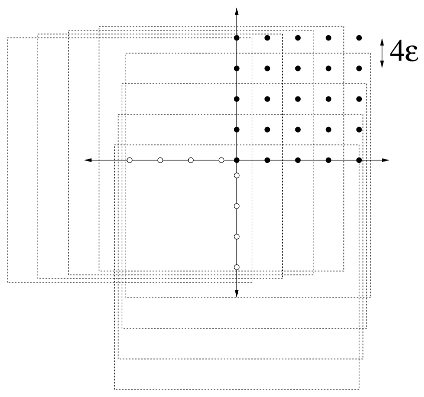

Figure 7: Lower bound for approximate range searching. Data points are shown in black. Ranges have been offset slightly for clarity.

Now consider any spanning path on the set of points P_1. From the construction it is easy to verify that every edge on this spanning path is stabbed by some query range in X. Since the number of edges on the spanning path is $\Omega(1/\epsilon^d)$ and $|X|$ is $O(1/\epsilon)$, it follows that the average number of edges stabbed by a query range in X is $\Omega(1/\epsilon^{d-1})$. Therefore there must exist a query range which stabs $\Omega(1/\epsilon^{d-1})$ edges. Thus the stabbing number of any spanning path exceeds this quantity. By Lemma 5, this is also a lower bound on the visiting number of any partition tree for P_1.

Let P_2 be any set of n distinct data points. We next show an $\Omega(\log n)$ lower bound on the visiting number of any partition tree T for P_2. Clearly, T must have a leaf at depth $\Omega(\log n)$. Let p be a data point in any such leaf. Let Q be an L_∞ ball centered at p. We choose the radius of Q to be sufficiently small to ensure that its $(1+\epsilon)$ expansion contains no other data point. It is easy to see that the range Q stabs the point sets corresponding to every proper ancestor of p. All such nodes are visited by the algorithm, hence this is also a lower bound on the visiting number of T.

Combining this with the results of the last paragraph, we have the following lower bound on the visiting number of any partition tree in the worst case. In fact, the lower bound holds even under the restriction to L_∞ balls.

Theorem 3 *For the set of query ranges consisting of balls in the L_∞ metric, the visiting number of any partition tree in the worst case is $\Omega(\log n + 1/\epsilon^{d-1})$.*

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 show the savings possible if one is willing to settle for approximations instead of requiring exact counts, 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. Rather than using the strict midpoint rule, we use a somewhat more flexible decomposition method, called the *fair-split rule* [2]. 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. Our decomposition process attempts to avoid centroid shrinking whenever it is not warranted. The reason is that there are optimizations that can be performed at splitting nodes that are not possible at shrinking nodes [2]. Our experience with the related approximate nearest neighbor searching problem has shown that, while these features are needed for highly clustered data sets, they are rarely needed in the sorts of naturally arising distributions we consider here. Throughout we used a bucket size (the maximum number of points associated with any leaf cell) of eight.

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. We present the following, since they were most representative.

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

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

Correlated Laplacian: An autoregressive source using the following recurrence to generate successive outputs $X_n = \rho X_{n-1} + W_n$, where W_n was chosen so that the marginal density of X_n is Laplacian with unit variance. The correlation coefficient ρ was taken to be 0.9. See [12] for more information.

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). 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 data points.

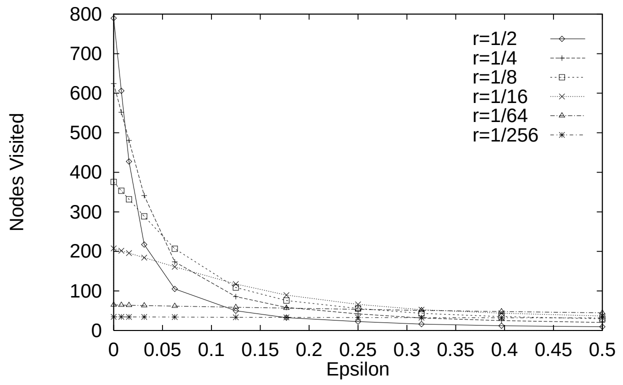

Figure 8: Number of nodes visited versus ϵ for the uniform distribution.

In Figures 8, 9, and 10, we show the average number of nodes visited as a function of ϵ, for each of the distributions. The number of data points is 65,536. Since the algorithm does a constant amount of work for each node visited, the number of nodes visited accurately reflects its running time. (We

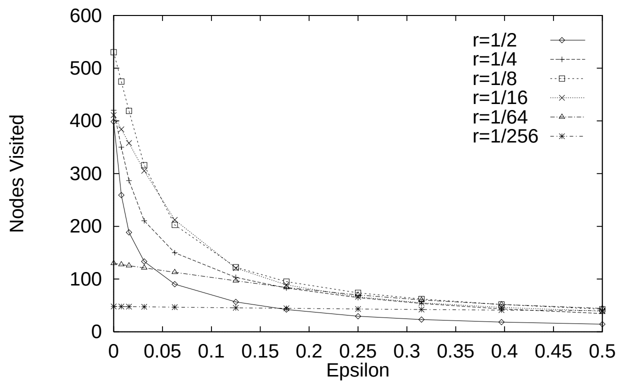

Figure 9: Number of nodes visited versus ϵ for the clustered Gaussian distribution.

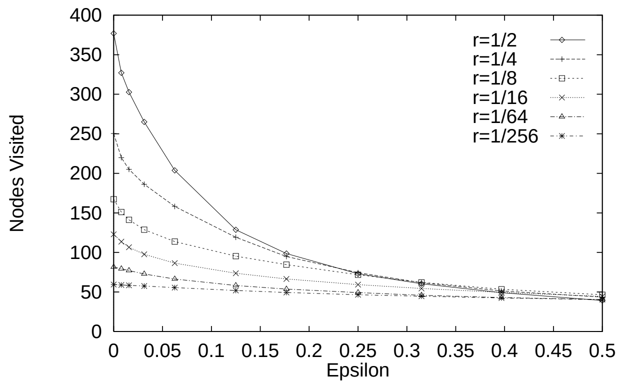

Figure 10: Number of nodes visited versus ϵ for the correlated Laplacian distribution.

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.) As predicted in our analysis, as ϵ increases from 0 to even a small value such as 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. In light of the results of this paper, the reason is obvious, namely, that the complexity of the approximate range searching problem grows logarithmically with n. In contrast the best known algorithms for the exact problem have running times that grow as $n^{1/2}$ in dimension 2 (even under the assumption of uniformly distributed data).

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.

We measured the actual *average error* committed by the algorithm, which is defined as follows. 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, $(r - r')/r$. Otherwise 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 do better than this. To see how much better it does on average, 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 *average error* of the query. If no points were eligible for misclassification, then this quantity is zero.

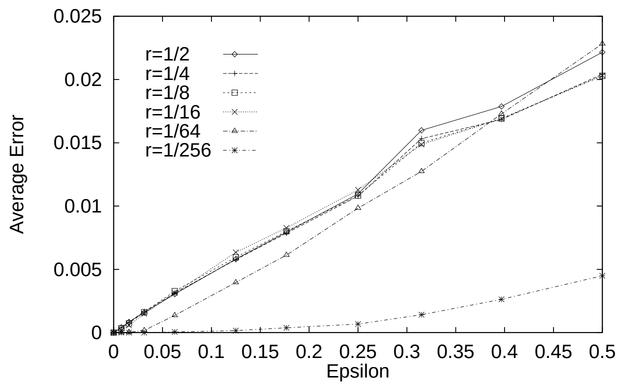

Figure 11: Average error versus ϵ for the uniform distribution.

In Figures 11, 12, and 13, we show the average error as a function of ϵ, for 65,536 data points. The key observation is that average error appears to

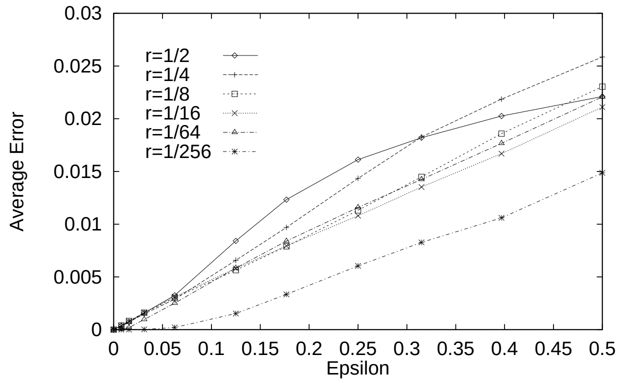

Figure 12: Average error versus ϵ for the clustered Gaussian distribution.

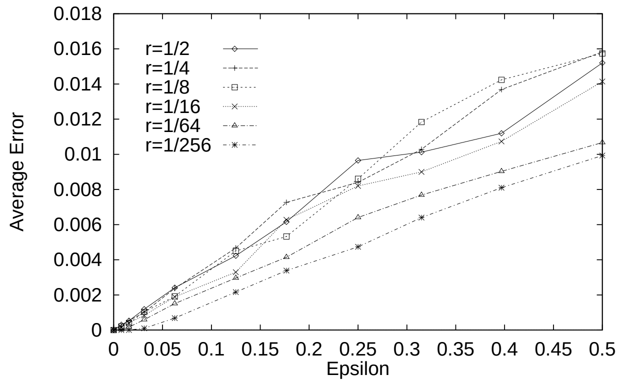

Figure 13: Average error versus ϵ for the correlated Laplacian distribution.

vary almost linearly with ϵ (depending on distribution, dimension, and other factors). In dimension 2, average errors were frequently less than 0.06ϵ, and in all distributions average error was never greater 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 average errors in the range of 1.5% to 3%.

6 Conclusions

We have presented a data structure for answering approximate range queries, where the error allowed by the algorithm is a function of the diameter of the range. We have shown that in any fixed dimension d, a set of n points in $\mathbf{R}^d$ 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, and for convex ranges the running time is $O(\log n + (1/\epsilon)^{d-1})$. We also presented a lower bound of $\Omega(\log n + (1/\epsilon)^{d-1})$ for approximate range searching based on the partition tree model. This implies that our algorithm is asymptotically optimal for convex ranges (assuming fixed dimensions). The algorithm is quite practical, and has been implemented.

There are a number of interesting open problems to be considered. The first involves the relatively high exponential dependence on dimension. Can these exponential factors be reduced, say down the lines of recent research in the area of approximate nearest neighbor searching? An intriguing question is the fact that the algorithm's observed average error tends to be much lower than the allowed error factor ϵ. Is there a good explanation for this phenomenon?

References

[1] S. Arya, D. M. Mount. Algorithms for fast vector quantization. In *Proc. of DCC '93: Data Compression Conference*, eds. J. A. Storer and M. Cohn, IEEE Press, pages 381–390, 1993.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45:891–923, 1998.

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

[4] M. Bern. Approximate closest-point queries in high dimensions. *Inform. Process. Lett.*, 45:95–99, 1993.

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

[6] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42:67–90, 1995.

[7] 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.

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

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

[10] 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.

[11] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, 1999.

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

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

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

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

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

[17] D. E. Willard. Polygon retrieval. *SIAM J. Comput.*, 11:149–165, 1982.

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