Expected-Case Complexity of Approximate Nearest Neighbor Searching *

Sunil Arya[†] Ho-Yam Addy Fu[‡]

Abstract

Most research in algorithms for geometric query problems has focused on their worst-case performance. But when information on the query distribution is available, the alternative paradigm of designing and analyzing algorithms from the perspective of expected-case performance appears more attractive. We study the approximate nearest neighbor problem from this point of view.

As a first step in this direction, we assume that the query points are chosen uniformly from a hypercube that encloses all the data points; however, we make no assumption on the distribution of data points. We investigate three simple variants of partition trees: sliding-midpoint, balance-split, and hybrid-split trees. We show that with these simple tree-based data structures, it is possible to achieve linear space and logarithmic or polylogarithmic query time in the expected case. In contrast, the data structures known to achieve linear space and logarithmic query time in the worst case are complex, and algorithms on them run more slowly in practice. Moreover, for the sliding-midpoint tree, we prove that it achieves optimal expected query time under reasonable assumptions.

1 Introduction

The main focus in the design of data structures and algorithms for geometric query problems has been to obtain optimal worst-case query time. However, in many applications, the average time it takes to answer a query is more important than the time it takes to answer a query in the worst case. Thus, assuming we have information on the query distribution, we believe it is prudent to incorporate it in the design with a view to minimize the *expected* query time or to provide simpler data structures. We study the approximate nearest neighbor problem from this perspective.

Nearest neighbor searching is a fundamental problem in computational geometry with applications in numerous areas such as pattern recognition [12], data compression [18], information retrieval [11] and multimedia databases [16]. Since the problem is very difficult to solve exactly in high dimensions, researchers have investigated the approximate nearest

*This research was supported in part by RGC CERG HKUST736/96E and RGC DAG96/97.EG40. A preliminary version of this paper is to appear in the *11th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2000.

[†]Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. Email: arya@cs.ust.hk

[‡]Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. Email: csaddy@cs.ust.hk

neighbor problem, defined as follows. Consider a set S of n data points in R^d. Given a query point q and an error bound $\epsilon > 0$, a point p is a $(1+\epsilon)$-approximate nearest neighbor, if $\frac{dist(p,q)}{dist(p^*,q)} \leq (1+\epsilon)$, where p^* is the true nearest neighbor of q. We assume that distances are measured using the Euclidean metric.

Without loss of generality, we may consider that the data points lie inside the unit hypercube U^d. Throughout we assume that the dimension d and the error bound ϵ are fixed constants, independent of the number n of data points. For our expected-case analysis, we assume that the query points are chosen from the uniform distribution in U^d, but make no assumption on the distribution of data points. While the assumption of uniformly distributed query points is admittedly simplistic, we think it is a natural first step towards the design of algorithms for more general query distributions. We design and analyze three data structures based on partition trees that employ only axis-orthogonal splitting planes. In each case, we use the *priority search* approach (devised by Arya et al. [5]) for finding the approximate nearest neighbor. The three partition trees we study use linear space and can be built in $O(n \log n)$ time using well-known techniques. The three tree structures are called the sliding-midpoint tree, balance-split tree, and hybrid-split tree. The sliding-midpoint tree was introduced by Mount and Arya [24], while the other two trees are novel. The main features of our data structures are:

- The partition trees we use are extremely simple. The cells are recursively subdivided using hyperplanes orthogonal to the coordinate-axes. In contrast, Arya et al. [5] and, more recently, Duncan et al. [13] have proposed partition trees that achieve $O(\log n)$ worst-case query time, but they both use complex ways of subdividing a cell. (In addition to axis-orthogonal splits, Arya et al. allow a *shrink* operation, which can generate a cell that is the set-theoretic difference of two rectangles. Duncan et al. restrict themselves to splits with a hyperplane; however, the hyperplane is not necessarily axis-orthogonal.) Although both partition trees provide optimal worst-case query time, the features introduced to ensure this property hurt their practical performance.

- The sliding-midpoint, hybrid-split, and balance-split trees achieve expected query times of $O(\log n), O(\log n \log\log n)$, and $O(\log^d n)$, respectively.

- The sliding-midpoint tree achieves *optimal* expected query time (ignoring constant factors) under reasonable assumptions. Note that this result, perhaps our most interesting one, holds irrespective of the data distribution. Moreover, the technique used in obtaining this result serves as a basis for the planar point-location results obtained in [1].

- The algorithms run fast in practice. It has been pointed out by Arya and Mount [2] that priority search can be performed most efficiently on partition trees that use axis-orthogonal splits. Briefly, the reason is that updating the distances to the children of a node can be carried out in constant time, independently of dimension, which is a big saving, especially in high dimension spaces.

- The sliding-midpoint and hybrid-split trees yield sublogarithmic (even constant) expected query time, when the data points are clustered in lower-dimensional subspaces.

- The worst-case query time obtained by the balance-split and hybrid-split trees is $O(\log^d n)$. For partition trees using axis-orthogonal hyperplanes, there are no previous results showing that sublinear worst-case query time is achievable.

Previous Work. In one dimension, the nearest neighbor problem can be solved with $O(n)$ space and $O(\log n)$ worst-case query time by using binary search on sorted data points. The same performance can also be achieved in two dimensions by constructing a Voronoi diagram and building a fast planar point-location data structure for it [6]. However, in dimension three or higher, the worst-case time complexity for the construction of the Voronoi diagram is $\Omega(n^{\lceil d/2 \rceil})$ and no method is known that simultaneously attains linear space and logarithmic query time in the worst case.

Much better worst-case algorithms are known for the approximate nearest neighbor problem in high dimensions. Bern [7] proposed a quadtree-based data structure that uses $O(n)$ space and provides $O(\log n)$ worst-case query time. However, the error bound ϵ for his algorithm is a fixed function of the dimension. Arya and Mount [3] presented a randomized algorithm that achieves polylogarithmic worst-case query time and almost linear space. In their algorithm, the error bound ϵ is an arbitrary positive constant, fixed at preprocessing time.

Arya et al. [5] gave an improved algorithm that provides $O((1/\epsilon)^d \log n)$ worst-case query time and uses $O(n)$ space. This worst-case query time is optimal (ignoring constant factors depending on d and ϵ) since it takes $\Omega(\log n)$ time to find the approximate nearest neighbor even in one dimension. Furthermore, once the data structure is built, their algorithm can answer queries for any value of ϵ. Their approach is based on constructing a partition tree with two important properties. First, the height of the tree is $O(\log n)$. Second, the cells associated with the nodes of the tree are *fat* (i.e., have bounded aspect ratio). Recently, Duncan et al. [13] have given another construction of a partition tree that has these two properties.

Clarkson [10] and later Chan [9] showed how to reduce the query time to $O((1/\epsilon)^{(d-1)/2} \log n)$, but it increased the space requirement to $O((1/\epsilon)^{(d-1)/2} n \log n)$. Kleinberg [20] provided an algorithm that has query time polynomial in $\log n$, d and ϵ but requires $O((n \log d)^{2d})$ space. Two groups of researchers, Kushilevitz et al. [21] and Indyk and Motwani [19], have obtained algorithms that eliminate exponential dependencies on dimension in both query time and space. The space required by their algorithms is polynomial in d and n, and the query time is polynomial in $\log n$ and d. The space and query time grow exponentially with $(1/\epsilon)$ but not with dimension.

2 Summary of Results

We study partition trees based on three different splitting rules, mainly from the perspective of expected-case query time (but we will also discuss their worst-case performance). We assume that d and ϵ are fixed constants, independent of n. However, we will show the dependence on ϵ in some of the asymptotic results.

Balance-split tree. The first subdivision is based on a new splitting rule called *balance-split*, which works as follows: Given a cell, it partitions it into two parts by a hyperplane orthogonal to the longest side. The hyperplane is located so that roughly half the data points lie on each side of it.

For the balance-split tree, our analysis shows that the worst-case query time is $O((1/\epsilon)^{d(d+1)/2} \log^d n)$, and the expected-case query time is $O((1/\epsilon)^d \log^d n)$. Note that the worst-case query time is $O(\log^d n)$ compared to $O(\log n)$ using the worst-case optimal partition trees [5, 13]. However, the main advantage of our data structure is its simplicity. Also, since it uses only axis-orthogonal splits, it should give better query times in practice.

Another way of viewing this result is the following. One of the most popular data structures used in nearest neighbor applications is the *kd-tree* devised by Friedman et al. [17]. This partition tree recursively splits the data set into two sets of equal size, using a hyperplane orthogonal to the dimension in which the points have maximum *spread* (difference of maximum and minimum coordinate). (See Figure 1.) Friedman et al. showed that it could be used to answer nearest neighbor queries in $O(n)$ space and $(\log n)$ expected time, under certain restrictive assumptions on both the data and the query distribution. One can easily construct examples in which their tree yields worst-case query time as bad as $\Omega(n)$. Our result shows that with a small modification to their rule —splitting the longest dimension instead of the dimension in which points have the maximum spread—it is possible to achieve worst-case polylogarithmic query time.

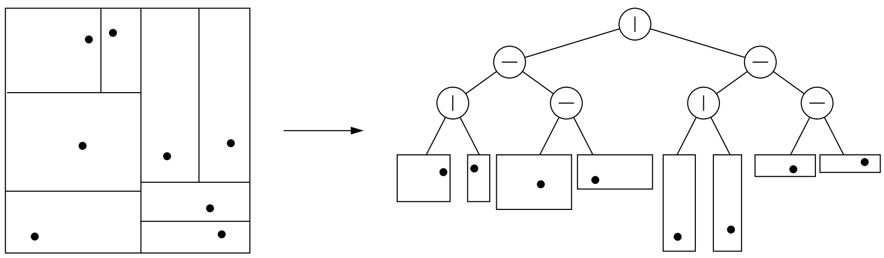

Figure 1: The kd-tree.

Sliding-midpoint tree. Our second result is for a partition tree based on a splitting rule called the *sliding-midpoint rule*, introduced by Mount and Arya [24]. This rule operates as follows: first, attempt to partition a cell by a hyperplane orthogonal to the longest side and located at its middle. If all the data points are on the same side of the hyperplane, then translate the hyperplane until it just passes through a data point. (See Figure 2.) While the worst-case query time for this tree can be as bad as $\Omega(n)$, Maneewongvatana and Mount have studied it empirically and observed that it performs well in practice [22]. We have two results pertaining to its expected query time, which give some theoretical justification for its good practical performance.

First, we prove that the expected-case query time for this tree is $O((1/\epsilon)^d \log n)$. Note that this is true irrespective of the data distribution. Second, we show that the sliding-midpoint tree is optimal in the following sense. Consider the class of algorithms obtained by running priority search on partition trees, where the splits are done by hyperplanes at ar-

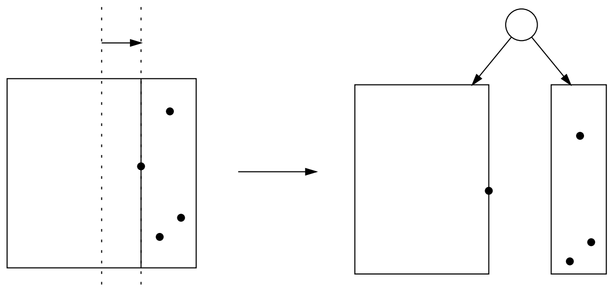

Figure 2: Sliding-midpoint rule.

bitrary orientations (i.e., not necessarily axis-orthogonal). We prove that, for any given data distribution, priority search on the sliding-midpoint tree achieves the minimum expected query time (up to a constant factor depending on d and ϵ) over all such algorithms. Our proof also implies that priority search on the sliding-midpoint tree attains lower expected query time compared to any algorithm that can be embodied as an algebraic decision tree using linear tests (i.e., involving the evaluation of a polynomial of degree one).

Hybrid-split tree. Finally, we study a partition tree built by applying the sliding-midpoint and balance-split rules alternately (we call this *hybrid-split*). Interestingly, the resulting tree combines the good properties of the balance-split and sliding-midpoint trees.

(i) It has $O((1/\epsilon)^{d(d+1)/2} \log^d n)$ worst-case query time. This is like the balance-split tree, but unlike the sliding-midpoint tree, which can have $\Omega(n)$ worst-case query time.

(ii) It has $O((1/\epsilon)^d \log n \log\log n + (1/\epsilon)^d \log(1/\epsilon) \log n)$ expected-case query time. This is like the sliding-midpoint tree (ignoring small factors), but unlike the balance-split tree, which can have $\Omega(\log^d n)$ expected-case query time.

The paper is organized as follows. In Section 4, we review the priority search approach proposed in [2, 5]. In Section 5, we describe the different splitting rules, including two new splitting rules, the balance-split rule and the hybrid-split rule. Section 6 sets up the framework used for the expected-case analysis of partition trees. In Sections 7, 8, 9, we analyze the worst-case and expected-case query time using the balance-split tree, the sliding-midpoint tree, and the hybrid-split tree, respectively. In Section 10, we establish the optimality of the sliding-midpoint tree, in the sense mentioned earlier. Finally, in Section 11, we present the experimental results.

3 Terminology

Let U^d denote the unit hypercube in d dimensions. Let S be a set of n data points in U^d (note that this is not a restriction since we can scale space to ensure this). Let T be a partition tree for S. Recall that each node of the partition tree is associated with a region of space, called a *cell*, and the subset of data points which lie in this cell. The cell associated with any node is partitioned into disjoint cells and associated with the children of the node.

The leaves of the tree contain at most a constant number of data points, called the *bucket size* (henceforth, assumed ≤ 1 for simplicity). In this paper, unless otherwise specified, we will assume that the cell associated with the root of the partition tree is the unit hypercube U^d.

Given a partition tree T, let $\mathcal{I}_T, \mathcal{L}_T$, and $\mathcal{N}_T$ denote the set of its internal nodes, leaf nodes, and all nodes, respectively. For a node x of T, let T_x denote the subtree rooted at x, and let v_x denote the volume of C_x.

Let T be a partition tree for S such that the cells for the children are obtained by splitting the parent's cell by an axis-orthogonal hyperplane. In this case the cells associated with the nodes of T are d-dimensional rectangles. For a node x, let R_x denote the rectangle associated with it. We define the *size* of a rectangle to be the length of its longest side, and define the *size* of a node x, denoted s_x, to be the size of R_x.

4 Background

We briefly review some features of the priority search approach for approximate nearest neighbor searching, proposed by Arya and Mount [2] and Arya et al. [5].

4.1 Query Algorithm

Define the *distance* between a point q and a cell to be the closest distance between q and any part of the cell. The algorithm is based on visiting leaf cells in order of increasing distance from the query point q. For each leaf cell visited, the algorithm computes the distance between q and the data points stored with the leaf, and updates p, the closest neighbor to q found so far, and r, the distance to it. The algorithm terminates when the distance between q and the leaf cell exceeds $r/(1+\epsilon)$, returning p as the answer. For example, in Figure 3, the leaf cells have been labeled in order of increasing distance from the query point q. The algorithm terminates on visiting cell 8, since its distance from the query point exceeds $r/(1+\epsilon)$ (the dotted circle shown).

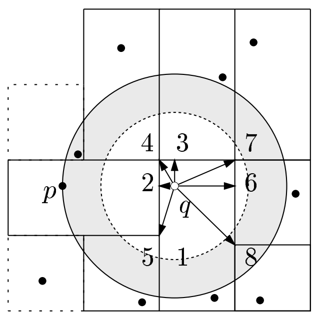

Figure 3: Algorithm overview.

To see that the algorithm works correctly for any partition tree, let B denote the ball of radius $r/(1+\epsilon)$ centered at q. Since the algorithm has already seen all leaf cells overlapping B, it is clear that B contains no data point. Thus, point p is a $(1+\epsilon)$-approximate nearest neighbor.

4.2 Distance Enumeration

In order to efficiently enumerate leaf nodes in increasing order of distance from the query point, Arya and Mount [2] use a priority queue to determine which node will be visited next. This method works for any partition tree.

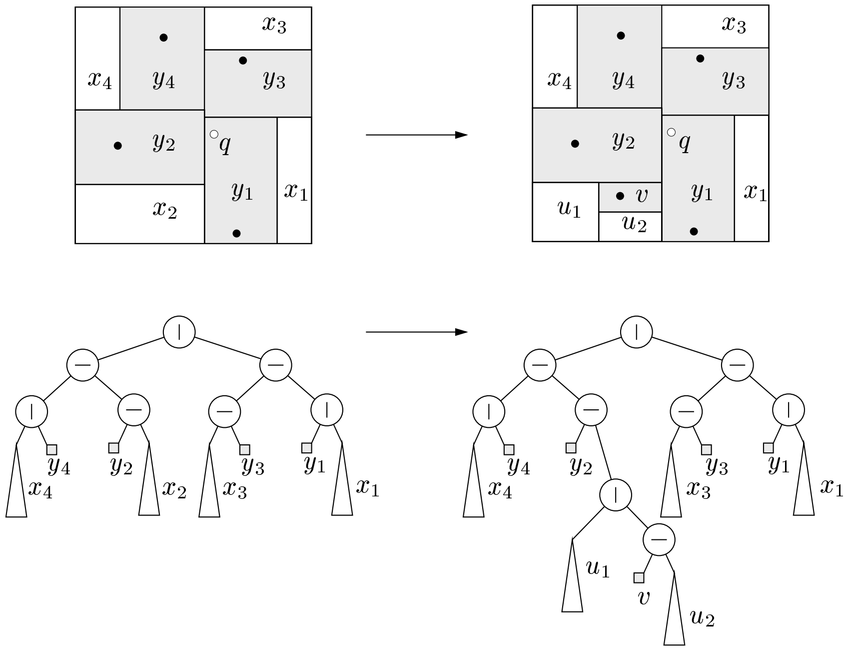

Figure 4: Priority Search.

Initially, the root of the tree is inserted into the heap. Then the following procedure is repeatedly carried out. First, the node with the highest priority is extracted from the heap, that is, the node which is closest to the query point. Then the algorithm descends the subtree associated with this node until reaching the leaf closest to the query point. For each internal node visited, the distance between the query point and the children of the node is computed. The child which is closer to the query point is visited recursively while the other child is inserted into the heap. When the algorithm reaches a leaf, it processes the data point associated with it. For example, in Figure 4, x_1, x_2, x_3, and x_4 are in the heap, and the closest node x_2 is extracted. When descending the tree to reach leaf v, nodes u_1 and u_2 are enqueued along the way. The correctness of the algorithm is based on the invariant that the set of leaves descended from the nodes in the priority queue are disjoint and their union is the set of all unvisited leaves.

4.3 Incremental Distance Computation

As mentioned in the introduction, priority search can be performed most efficiently on partition trees that use axis-orthogonal splits. The reason is as follows. Recall that for each internal node visited, the algorithm needs to compute the distance between the query point q and the children of the node. If we do not restrict ourselves to axis-orthogonal splits, this computation can take $\Theta(d)$ or even greater time. In contrast, for a node that is partitioned by an axis-orthogonal hyperplane, this computation can be done in $O(1)$ time *independent* of dimension d, using a technique called *incremental distance computation* [2]. From a practical perspective, the difference is significant especially in high dimensions.

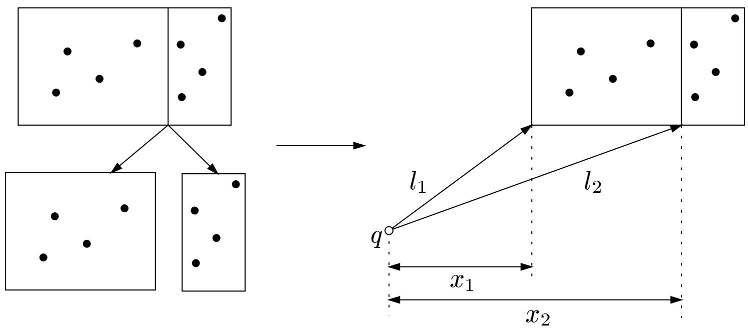

Figure 5: Incremental distance computation: $l_2^2 = l_1^2 - x_1^2 + x_2^2$.

Figure 5 illustrates the method in two dimensions. To avoid computing square roots, we compute the square of the distances. Suppose that we know the squared distance, as well as the distance along each dimension, from the query point q to the parent cell. Then for the child closer to q (the left one in the figure), the squared distance is the same as the squared distance to its parent. For the other child, since it differs from its parent in only one dimension, we can update its squared distance using a constant number of arithmetic operations.

4.4 Query Time

The analysis of the query time given in [5] employs the following two lemmas, which will also be useful for us. Lemma 4.1 implies that all the leaf cells visited, except possibly the last one, are large relative to their distance from the query point. Lemma 4.2 relates the query time to the number of internal nodes and leaf nodes visited. Their proofs follow from the discussion in [5]; we repeat them here for the sake of completeness.

Lemma 4.1 *([5]) Let T be a partition tree built using axis-orthogonal splits, such that each leaf cell contains a data point. Let q be a query point. Let r be the distance from q to the last leaf cell visited by priority search that does not cause the algorithm to terminate. Then the size of any leaf cell visited until then is $\geq r\epsilon/d$.*

Proof Suppose that priority search visits a leaf cell x of size $< r\epsilon/d$. Clearly, the diameter of x is then $< r\epsilon$ and so it contains a data point at distance $< r(1+\epsilon)$ from q. Thus the distance to the closest point seen by the algorithm would be $< r(1+\epsilon)$, implying that the leaf cell x meets the termination condition, which is a contradiction. □

Lemma 4.2 *([2, 5]) Let T be a partition tree built using axis-orthogonal splits. Given a query point q, and error bound ϵ, let I and L be the number of internal nodes and leaf nodes,* *respectively, visited by priority search. Then the query time is $O(I + Ld + L \log I)$, where the constant factor in the O-notation is independent of d and ϵ.*

Proof Each internal node does the following tasks. First, it updates the distance to its two children; since the children are created by an axis-orthogonal split, this can be done in $O(1)$ time using incremental distance computation. Then it inserts the farther child into the heap; using Fibonacci heaps, the amortized time for each insertion is $O(1)$. Finally, it descends to the closer child which takes $O(1)$ time.

Each leaf node does two tasks. First, it computes the distance between q and the data points stored with the leaf. Since each leaf has at most a constant number of data points, this takes $O(d)$ time. Second, it extracts the closest node from the heap. Since each internal node visited inserts one child into the heap, the size of the heap is at most I. Thus, it takes $O(\log I)$ time to extract the closest node. The time for each leaf node is therefore $O(d + \log I)$.

Initially the algorithm computes the distance between q and the root node, which takes an additional $O(d)$ time. The total query time is therefore $O(I + L(d + \log I) + d) = O(I + Ld + L \log I)$. $\square$

5 Splitting Rules

In this section, we describe the different splitting rules used to build data structures for approximate nearest neighbor searching. These rules include the standard kd-tree splitting rule [17], the midpoint-split rule [5], and the sliding-midpoint rule [22, 24]. In addition, we describe two new splitting rules, the balance-split rule and the hybrid-split rule, introduced by us.

Standard kd-tree splitting rule: The dimension along which the points have the maximum spread is chosen to be the splitting dimension. The splitting hyperplane is located orthogonal to the splitting dimension such that the number of data points on each side of it is nearly equal. (See Figure 1.) This rule ensures that the height of the tree is $O(\log n)$. However, the cells are not guaranteed to have aspect ratio (ratio of the longest to shortest side) bounded by a constant, and there is no good guarantee on the worst-case query time. However, under certain assumptions on the data and query distribution, Friedman et al. [17] have shown that the expected query time is $O(\log n)$.

Midpoint-split rule: The splitting hyperplane of this rule is located orthogonal to the longest side of the cell (in case of ties, any side may be picked), and at its middle. This rule may yield trees of arbitrary height. (See Figure 6.) Further, this rule may generate empty cells, so the size of the tree can also be arbitrarily large.

Sliding-midpoint rule: The sliding-midpoint rule is designed to overcome the unbounded size of the tree created by the midpoint-split rule. This rule first chooses a splitting hyperplane based on the midpoint-split rule, i.e., the hyperplane is placed orthogonal to the longest side of the cell at its middle. It chooses this hyperplane to partition

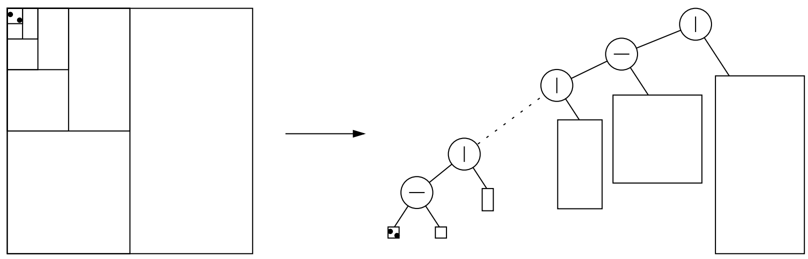

Figure 6: Midpoint-split rule: large size tree.

the cell if there are data points located on both sides. This means that the sliding-midpoint rule acts just like the midpoint-split rule if no empty leaf is created by the split. Otherwise, if all the data points lie on one side of the splitting hyperplane, it slides the hyperplane until it just passes through a point. The point which touches the splitting plane is assigned to the side which is originally empty. (See Figure 7.) Since the sliding-midpoint rule does not create any empty cell, the space used by the tree is $O(n)$. However, the rule does not try to balance the number of points among the children and its height can be as bad as $\Omega(n)$. We will still be able to show in Section 8 that the expected query time of the sliding-midpoint rule is $O(\log n)$.

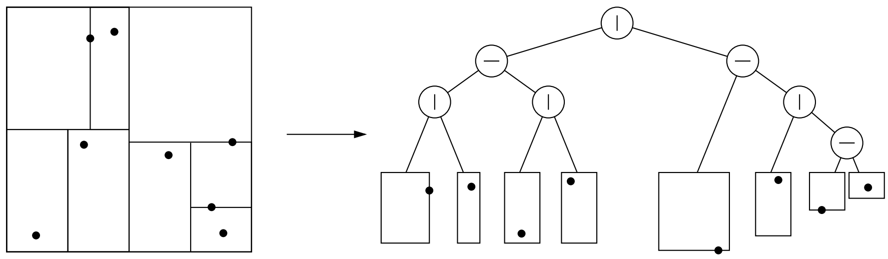

Figure 7: Sliding-midpoint tree.

Balance-split rule: The balance-split rule is similar to the standard kd-tree rule. The main difference is in the choice of the splitting dimension. While the standard kd-tree rule chooses the splitting dimension to be the dimension along which points have maximum spread, the balance-split rule chooses the splitting dimension to be the longest side of the cell. The splitting hyperplane is selected such that the number of data points on each side of the hyperplane is roughly equal. (See Figure 8.) We will show that the worst-case query time using the balance-split tree is $O(\log^d n)$.

Hybrid-split rule: The hybrid-split rule alternates between applying the sliding-midpoint rule and the balance-split rule at different levels of the tree. (See Figure 9.) The motivation for this rule comes from the fact that trees built using the sliding-midpoint rule yield low expected query time. However, the height of the sliding-midpoint tree and hence the worst-case query time can be as bad as $\Omega(n)$. By alternating between the sliding-midpoint rule and the balance-split rule, the height of the tree is guaranteed to be $O(\log n)$. Indeed, as we will show in Section 9, the resulting tree yields a worst-

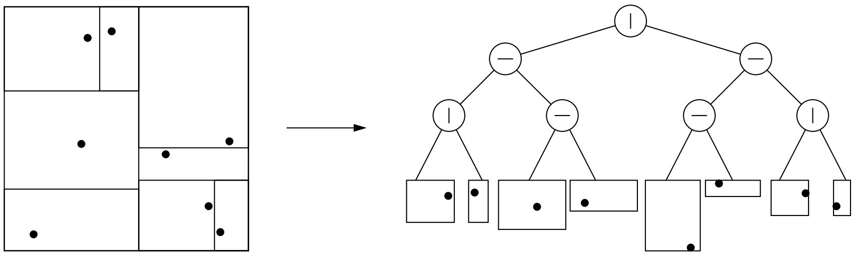

Figure 8: Balance-split rule.

case query time as good as the balance-split tree, and its expected query time is almost as good as the sliding-midpoint tree.

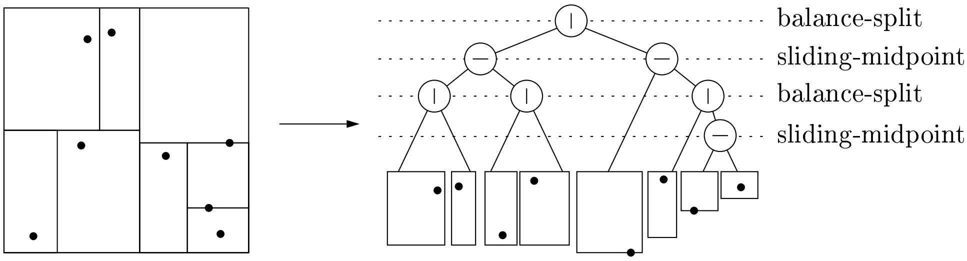

Figure 9: Hybrid-split rule.

6 Expected-Case Analysis: General Framework

In this section, we set up the basic framework for analyzing the expected query time of priority search on a hierarchical subdivision. We assume that the query point is chosen from the uniform distribution in U^d, the unit hypercube containing all the data points.

Before presenting the analysis, we need to make a small but important modification to priority search.

6.1 Modified Priority Search

First, we give some intuition for why this modification is needed. Recall from Section 4 that priority search is given a chance to terminate the search only at leaf nodes. Further, it follows from Lemma 4.1 that it sees at most one leaf cell of size smaller than $r\epsilon/d$, where r is the distance from the query point to the leaf cell (because if it sees such a leaf cell, it immediately causes the algorithm to terminate). However, while descending the tree to this small leaf cell, the algorithm may see a large number of small internal nodes. This is undesirable as it increases the query time, and prevents us from proving good bounds on the expected query time. Thus, the idea of the modifications is to allow priority search to terminate the search at internal nodes as well, so as to ensure that it sees at most one node (leaf or internal) of size $< r\epsilon/d$. This fact will prove to be important for some of our results.

The details of the modification are as follows. With each internal node x, we store its size s_x and a pointer to any data point p_x inside R_x. (Asymptotically, this does not increase the space or preprocessing requirements.) The search algorithm works in exactly the same way if a leaf node is visited. If an internal node x is visited, then let r denote its distance from the query point. If $s_x \geq r\epsilon/d$, the algorithm works just as before. However, if $s_x < r\epsilon/d$, the algorithm computes the distance from q to the associated data point p_x, updates the closest point seen so far, and terminates.

It is easy to see that the modified algorithm remains correct. Obviously, if the algorithm terminates at a leaf node, then it behaves exactly as the unmodified algorithm and is therefore correct. If it terminates at an internal node x at distance r from the query point q, then the diameter of R_x is at most $r\epsilon$, and so the associated data point p_x is at distance at most $r(1+\epsilon)$ from q. If the algorithm has not seen any data point within distance r from q, then point p_x is a $(1+\epsilon)$-approximate nearest neighbor. If the algorithm has already seen a data point u at distance $< r$, then u is the nearest neighbor. In either case, the algorithm returns the correct answer.

For the rest of this paper, we shall assume that priority search has been modified as indicated above. We note that Lemmas 4.1 and 4.2 continue to hold after this modification [5].

6.2 Upper Bound on Expected Number of Nodes Visited

In Lemma 6.1 we establish upper bounds on the expected number of internal nodes and leaf nodes visited by priority search, which depend only on the sizes of the rectangles associated with the nodes of the partition tree. These upper bounds are central to the expected-case analysis of trees based on specific splitting rules, given in Sections 7, 8, 9, and 10.

Lemma 6.1 *Let d and ϵ be any fixed constants. Let S be a set of n data points in U^d. Let T be a partition tree for S built using axis-orthogonal splits, such that each leaf cell contains a data point. Assume that the query point is chosen from the uniform distribution in U^d. Then*

(i) the expected number of internal nodes visited by priority search is $\leq 1 + (1 + 2d/\epsilon)^d \sum_{x \in \mathcal{I}_T} s_x^d$, and

(ii) the expected number of leaf nodes visited by priority search is $\leq 1+(1+2d/\epsilon)^d \sum_{x \in \mathcal{L}_T} s_x^d$.

Proof Let q be a query point. Let x be any node visited that does not cause the algorithm to terminate and let r denote the distance between q and R_x. We claim that $s_x \geq r\epsilon/d$. If x is an internal node, this follows in view of the modifications made to priority search, and if x is a leaf node, this follows from Lemma 4.1. Thus $r \leq s_x d/\epsilon$. That is, if a node x is visited and does not cause the algorithm to terminate, then the query point q must be at distance $\leq s_x d/\epsilon$ from R_x.

Recall that the query point is chosen from the uniform distribution in U^d. With each internal node x, associate a random variable Y_x which takes the value 1 if node x is visited and does not cause the algorithm to terminate, otherwise it takes the value 0. Let I denote the number of internal nodes visited by priority search. Clearly $I \leq 1 + \sum_{x \in \mathcal{I}_T} Y_x$. Here we have added 1 to take into account the last internal node visited which may have caused the algorithm to terminate. By linearity of expectation,

$$E[I] \leq 1 + \sum_{x \in \mathcal{I}_T} E[Y_x]. \tag{1}$$

Note that $E[Y_x]$ equals the probability that Y_x is 1. We compute an upper bound on this probability. From our earlier observation, if node x is visited and does not cause the algorithm to terminate, then q is at distance $\leq s_x d/\epsilon$ from R_x. Thus q lies inside the hypercube of size $(1 + 2d/\epsilon)s_x$ whose center coincides with the center of R_x. The volume of this hypercube, $(1 + 2d/\epsilon)^d s_x^d$, is therefore an upper bound on the desired probability, and hence on $E[Y_x]$. Substituting in Eq. (1) proves (i). The proof of (ii) is similar and is omitted. □

The upper bounds given by Lemma 6.1 are not necessarily tight. However, they prove to be very useful in our analysis of partition trees formed by different splitting rules. In each case, our approach will be to first compute $\sum_{x \in \mathcal{I}_T} s_x^d$ and $\sum_{x \in \mathcal{L}_T} s_x^d$ for the partition tree T, in terms of the number of data points n. We then use Lemmas 6.1 and 4.2 to bound the expected query time.

7 The Balance-Split Tree

In this section, we analyze the worst-case and expected-case query time of priority search on the balance-split tree. We prove that, for any fixed dimension d and error bound ϵ, the balance-split tree yields worst-case query time of $O((1/\epsilon)^{d(d+1)/2} \log^d n)$, and expected-case query time of $O((1/\epsilon)^d \log^d n)$.

7.1 Worst-Case Analysis

We present a lemma, which computes an upper bound on the number of large leaf cells that can overlap a ball of radius r. Using this lemma, we bound the worst-case query time in Theorem 7.1.

Lemma 7.1 *Let S be a set of n data points in U^d. Let T be any partition tree for S of $O(\log n)$ height, with the property that the child rectangles are formed by splitting orthogonal to the longest side of the parent rectangle. The number of leaf cells of size $\geq r\epsilon/d$ that overlap a ball of radius r is at most $O((1/\epsilon)^{d(d+1)/2} \log^{d-1} n)$. The constant factor in the O-notation depends on d.*

Proof Define a node x to be a k-type node if it has exactly k sides of length $< r\epsilon/d$ (i.e., the remaining $d - k$ sides have length $\geq r\epsilon/d$). Figure 10 shows an example in two dimensions.

Let B denote the given ball of radius r. The proof works by counting the number of k-type leaves of T that overlap B, for $0 \leq k \leq d-1$, and summing this over all k.

Recall that the root of T is associated with the unit hypercube, and so its type is either 0 or d. If its type is d, then its size is $< r\epsilon/d$, and there can be no leaf cells of size $\geq r\epsilon/d$

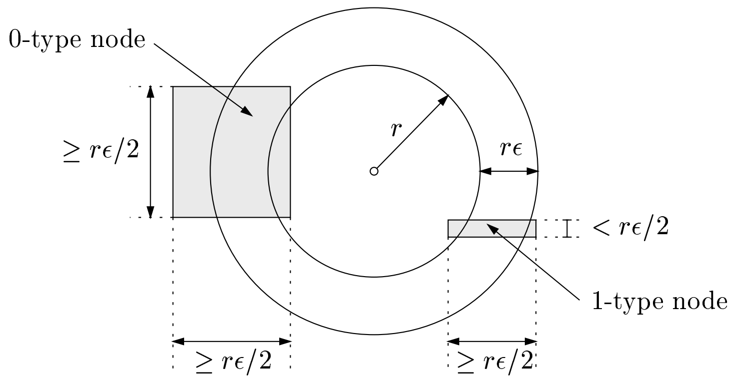

Figure 10: Analysis of the balance-split rule: 0-type and 1-type nodes.

that overlap B. Thus, the lemma is trivially true. So assume that the root of T is of 0-type. Since we split exactly one side of a rectangle to form its two children, the type of a node either remains the same or increases by at most one as we descend one level in the tree. (We call this the *monotonicity* property.)

We claim that the number of k-type leaves and nodes that overlap B is $O((1/\epsilon)^{(d-k/2)(k+1)} \log^k n)$ and $O((1/\epsilon)^{(d-k/2)(k+1)} \log^{k+1} n)$, respectively. The proof is by induction on the type of the node.

For the basis, we count the number of 0-type nodes that overlap B. Remove all nodes from T that do not overlap B; let T' be the tree consisting of the remaining nodes. Consider all nodes in T' that are of 0-type. By the monotonicity property, these must form a subtree T'', which includes the root of T'. Since the rectangles associated with the leaves of T'' are disjoint and each side of these rectangles has length $\geq r\epsilon/d$, the number of leaves in T'' is at most the number of d-dimensional hypercubes of length $r\epsilon/d$ that can overlap a ball of radius r, which by a simple packing argument is $O((1/\epsilon)^d)$. Since the height of T'' is $O(\log n)$, the number of nodes in T'' is $O((1/\epsilon)^d \log n)$. Thus, the number of 0-type leaves and 0-type nodes that overlap B is $O((1/\epsilon)^d)$ and $O((1/\epsilon)^d \log n)$, respectively.

For the induction hypothesis, assume that the claim is true for number of leaves and nodes of type $\leq k$, where $k < d-1$. We will prove the claim for number of $(k+1)$-type leaves and $(k+1)$-type nodes.

Consider all nodes in T' that are of $(k+1)$-type; these form a collection of maximal subtrees. First, we calculate the number of such subtrees. Let T'' be any such subtree and let x denote the root of T''. By the monotonicity property, the parent of x in T' must be of type k. Since T' is a binary tree, the number of subtrees is no more than twice the number of k-type nodes in T'. By the induction hypothesis, it follows that the number of subtrees is $O((1/\epsilon)^{(d-k/2)(k+1)} \log^{k+1} n$

Next we calculate an upper bound on the number of leaves and nodes in T''. For any node y, define $D_y \subseteq \{1, 2, \ldots d\}$ to be the set of dimensions for which the corresponding sides of R_y have length $< r\epsilon/d$. Let z be any node in T''. Since the length of the sides of R_z can be no more than the length of the corresponding sides of R_x, and nodes z and x have exactly $k+1$ sides of length $< r\epsilon/d$, it follows that $D_z = D_x$. Since the balance-split rule always splits the longest side of the rectangle, it is clear that the rectangles associated with the leaves of T'' are obtained from R_x by splitting orthogonal to the dimensions not in D_x. Project B and the rectangles associated with the leaves of T'' on the set of dimensions not in D_x. Clearly, in the projected space, the rectangles are disjoint and each side of the rectangle has length $\geq r\epsilon/d$. Thus, the number of leaves in T'' is no more than the number of $(d-k-1)$-dimensional hypercubes of length $r\epsilon/d$ that can overlap a $(d-k-1)$-dimensional ball of radius r, which is $O((1/\epsilon)^{d-k-1})$. Also, since the height of T'' is $O(\log n)$, the number of nodes in T'' is $O((1/\epsilon)^{d-k-1}\log n)$.

Since the number of subtrees is $O((1/\epsilon)^{(d-k/2)(k+1)}\log^{k+1} n)$, and the number of leaves in each subtree is $O((1/\epsilon)^{d-k-1})$, the total number of $(k+1)$-type leaves that overlap B is $O((1/\epsilon)^{(d-k/2)(k+1)+(d-k-1)}\log^{k+1} n)$ which simplifies to $O((1/\epsilon)^{(d-(k+1)/2)(k+2)}\log^{k+1} n)$. Also, since number of nodes in each subtree is $O((1/\epsilon)^{d-k-1}\log n)$, the total number of $(k+1)$-type nodes that overlap B is $O((1/\epsilon)^{(d-(k+1)/2)(k+2)}\log^{k+2} n)$. This completes the proof by induction.

It remains to sum the number of leaves and nodes of different types that overlap B. It is easy to see that this summation is dominated by the number of $(d-1)$-type leaves and $(d-1)$-type nodes. This completes the proof of the lemma. □

Theorem 7.1 *Let S be a set of n data points in U^d. Let T be any partition tree for S of $O(\log n)$ height, with the property that the child rectangles are formed by splitting orthogonal to the longest side of the parent rectangle. Given any ϵ, the worst-case query time of priority search on T is $O((1/\epsilon)^{d(d+1)/2}\log^d n)$. The constant factor in the O-notation depends on d.*

Proof Let q be a query point. Let I and L denote the number of internal and leaf nodes visited by priority search. Let r be the distance to the last leaf cell visited, which does not cause the algorithm to terminate. By Lemma 4.1, any leaf cell visited until then has a size of at least $r\epsilon/d$. Obviously all these leaf cells overlap the ball of radius r centered at q, since the algorithm visits leaf cells in increasing order of distance from q. Thus, by Lemma 7.1, $L = O((1/\epsilon)^{d(d+1)/2}\log^{d-1} n)$. Since the height of the tree is $O(\log n)$, it follows from the discussion in Section 4.2 that the number of internal nodes visited is $O(L\log n)$. Therefore, by Lemma 4.2, the worst-case query time is $O(L\log n)$. Using the bound on L, the desired result follows. □

Since the tree generated by the balance-split rule satisfies the condition of Theorem 7.1, we have the following corollary.

Corollary 7.1 *Let S be a set of n data points in U^d. Let T be the partition tree for S built using the balance-split rule. Given any ϵ, the worst-case query time of priority search on T is $O((1/\epsilon)^{d(d+1)/2}\log^d n)$. The constant factor in the O-notation depends on d.*

7.2 Expected-Case Analysis

Obviously, the upper bound given by Corollary 7.1 on the worst-case query time applies to the expected-case query time as well. In this subsection, we prove that the expected-case query time is $O((1/\epsilon)^d\log^d n)$, which is much better in terms of the dependence on ϵ.

Recall from Section 6 that the expected query time for a partition tree T is closely related to the quantities $\sum_{x \in \mathcal{I}_T} s_x^d$ and $\sum_{x \in \mathcal{L}_T} s_x^d$. In Lemmas 7.2, 7.3, and 7.4, we compute an upper bound on these two quantities in terms of n, for the balance-split tree. Finally in Theorem 7.2 we use these bounds to derive an upper bound on the expected query time.

Lemma 7.2 *Let R be a rectangle in d dimensions. Consider that we split R by a hyperplane orthogonal to its longest side; let R_1 and R_2 denote the resulting rectangles. For $1 \le i \le d$, let $l_i, l_{1,i}$, and $l_{2,i}$ denote the length of the ith longest side of R, R_1, and R_2, respectively. For any i, $1 \le i < d$,*

$$\left(\prod_{m=1}^{i-1} l_{1,m}\right) l_{1,i}^{d-i+1} + \left(\prod_{m=1}^{i-1} l_{2,m}\right) l_{2,i}^{d-i+1} \le \left(\prod_{m=1}^{i-1} l_m\right) l_i^{d-i+1} + \left(\prod_{m=2}^{i} l_m\right) l_{i+1}^{d-i+1}. \quad (2)$$

Proof Let j be the side of R that is split, and let l_1' and l_1'' denote the length of the corresponding side of rectangles R_1 and R_2, respectively. Clearly, $l_1 = l_1' + l_1''$. We consider three cases.

Case 1: l_1' is the length of one of i longest sides of R_1 and l_1'' is the length of one of the i longest sides of R_2.

Thus, the set of the lengths of the i longest sides of R_1 is $\{l_1', l_2, l_3, \ldots, l_i\}$, and the set of the lengths of the i longest sides of R_2 is $\{l_1'', l_2, l_3, \ldots, l_i\}$. Clearly, $l_{1,i} \le l_i$ and $l_{2,i} \le l_i$. It follows that

$$\begin{aligned} \left(\prod_{m=1}^{i-1} l_{1,m}\right) l_{1,i}^{d-i+1} &= \left(\prod_{m=1}^{i} l_{1,m}\right) l_{1,i}^{d-i} \\ &\le l_1' l_2 l_3 \ldots l_i l_i^{d-i} = l_1' l_2 l_3 \ldots l_{i-1} l_i^{d-i+1}. \end{aligned} \quad (3)$$

Similarly,

$$\left(\prod_{m=1}^{i-1} l_{2,m}\right) l_{2,i}^{d-i+1} \le l_1'' l_2 l_3 \ldots l_{i-1} l_i^{d-i+1}. \quad (4)$$

Summing up Eqs. (3) and (4),

$$\begin{aligned} \left(\prod_{m=1}^{i-1} l_{1,m}\right) l_{1,i}^{d-i+1} + \left(\prod_{m=1}^{i-1} l_{2,m}\right) l_{2,i}^{d-i+1} &\le (l_1' + l_1'') l_2 \ldots l_{i-1} l_i^{d-i+1} \\ &= l_1 l_2 \ldots l_{i-1} l_i^{d-i+1}. \end{aligned}$$

Note that this is the first term of the right hand side of Eq. (2), and so the lemma holds in this case.

Case 2: l_1' is not the length of one of the i longest sides of R_1.

Thus, the lengths of the i longest sides of R_1 in decreasing order are $l_2 \le l_3 \le \ldots \le l_{i+1}$. That is, $l_{1,m} = l_{m+1}$, for $1 \le m \le i$. Therefore

$$\left(\prod_{m=1}^{i-1} l_{1,m}\right) l_{1,i}^{d-i+1} = \left(\prod_{m=1}^{i-1} l_{m+1}\right) l_{i+1}^{d-i+1}. \quad (5)$$

Since rectangle R_2 is contained in rectangle R, it follows that $l_{2,m} \leq l_m$, $1 \leq m \leq d$. Thus

$$\left(\prod_{m=1}^{i-1} l_{2,m}\right) l_{2,i}^{d-i+1} \leq \left(\prod_{m=1}^{i-1} l_m\right) l_i^{d-i+1}. \tag{6}$$

Summing up Eqs. (5) and (6) proves Eq. (2).

Case 3: $l_1^{''}$ is not the length of one of the i longest sides of R_2.

This case is similar to Case 2, and so we omit it. $\square$

Lemma 7.3 *Let S be a set of n data points in U^d. Let T be any partition tree for S with the property that the child rectangles are formed by splitting orthogonal to the longest side of the parent rectangle. Let x be any node of T. For $1 \leq i \leq d$, let l_i denote the ith longest side of R_x. Let $\mathcal{M}$ denote any set of pairwise disjoint nodes at levels $\leq w$ in the subtree rooted at x (i.e., no node is an ancestor of the other). Then*

$$\sum_{y \in \mathcal{M}} s_y^d \leq \sum_{j=1}^{d} \left(\left(\prod_{i=1}^{j-1} l_i \right) l_j^{d-j+1} (w-1)^{j-1} \right). \tag{7}$$

Proof The proof is by induction on w. For the basis, observe that when $w = 1$, both the left hand side and right hand side of Eq. (7) equal l_1^d.

For the induction hypothesis, assume that the claim is true for $w < k$. We will prove the claim for $w = k$. If x has no children, or if $\mathcal{M}$ consists only of the node x, the claim is true by the induction hypothesis. So assume that x has two children x_1 and x_2, and let $\mathcal{M}_1$ and $\mathcal{M}_2$ be the nodes in $\mathcal{M}$ belonging to the subtrees rooted at x_1 and x_2, respectively. Let $l_{1,i}$ and $l_{2,i}$ denote the ith longest sides of R_{x_1} and R_{x_2}, respectively. Clearly,

$$\sum_{x \in \mathcal{M}} s_x^d = \sum_{x \in \mathcal{M}_1} s_x^d + \sum_{x \in \mathcal{M}_2} s_x^d.$$

Applying the induction hypothesis to nodes x_1 and x_2,

$$\begin{aligned}
\sum_{y \in \mathcal{M}} s_y^d &\leq \sum_{j=1}^{d} \left(\prod_{i=1}^{j-1} l_{1,i} \right) l_{1,j}^{d-j+1} (w-2)^{j-1} + \sum_{j=1}^{d} \left(\prod_{i=1}^{j-1} l_{2,i} \right) l_{2,j}^{d-j+1} (w-2)^{j-1} \\
&= \sum_{j=1}^{d} \left(\left(\prod_{i=1}^{j-1} l_{1,i} \right) l_{1,j}^{d-j+1} + \left(\prod_{i=1}^{j-1} l_{2,i} \right) l_{2,j}^{d-j+1} \right) (w-2)^{j-1}.
\end{aligned}$$

Since $(w-2)^{j-1} \leq (w-1)^{j-2}(w-2) = (w-1)^{j-1} - (w-1)^{j-2}$, this can be written as

$$\begin{aligned}
\sum_{y \in \mathcal{M}} s_y^d &\leq l_{1,1}^d + l_{2,1}^d \\
&\quad + \sum_{j=2}^{d} \left(\left(\prod_{i=1}^{j-1} l_{1,i} \right) l_{1,j}^{d-j+1} + \left(\prod_{i=1}^{j-1} l_{2,i} \right) l_{2,j}^{d-j+1} \right) \left((w-1)^{j-1} - (w-1)^{j-2} \right).
\end{aligned}$$

Using Lemma 7.2, and the fact that $\prod_{i=1}^{d} l_{1,i} + \prod_{i=1}^{d} l_{2,i} = \prod_{i=1}^{d} l_i$ (since $\prod_{i=1}^{d} l_{1,i} = v_{x_1}, \prod_{i=1}^{d} l_{2,i} = v_{x_2}, \prod_{i=1}^{d} l_i = v_x$, and $v_{x_1} + v_{x_2} = v_x$),

$$\begin{aligned}
\sum_{y \in \mathcal{M}} s_y^d &\leq l_1^d + l_2^d + \left(\prod_{i=1}^{d} l_i\right)\left((w-1)^{d-1} - (w-1)^{d-2}\right) \\
&\quad + \sum_{j=2}^{d-1}\left(\left(\left(\prod_{i=1}^{j-1} l_i\right) l_j^{d-j+1} + \left(\prod_{i=1}^{j-1} l_{i+1}\right) l_{j+1}^{d-j+1}\right)\left((w-1)^{j-1} - (w-1)^{j-2}\right)\right) \\
&\leq l_1^d + l_2^d + \left(\left(\prod_{i=1}^{d} l_i\right)(w-1)^{d-1} - \left(\prod_{i=1}^{d} l_i\right)(w-1)^{d-2}\right) \\
&\quad + \sum_{j=2}^{d-1}\left(\left(\left(\prod_{i=1}^{j-1} l_i\right) l_j^{d-j+1} + \left(\prod_{i=1}^{j-1} l_{i+1}\right) l_{j+1}^{d-j+1}\right)(w-1)^{j-1}\right) \\
&\quad - \sum_{j=2}^{d-1}\left(\left(\prod_{i=1}^{j-1} l_i\right) l_j^{d-j+1}(w-1)^{j-2}\right).
\end{aligned}$$

Regrouping the terms and simplifying,

$$\begin{aligned}
\sum_{y \in \mathcal{M}} s_y^d &\leq \sum_{j=1}^{d}\left(\left(\prod_{i=1}^{j-1} l_i\right) l_j^{d-j+1}(w-1)^{j-1}\right) \\
&\quad + \sum_{j=1}^{d-1}\left(\left(\prod_{i=1}^{j-1} l_{i+1}\right) l_{j+1}^{d-j+1}(w-1)^{j-1}\right) - \sum_{j=2}^{d}\left(\left(\prod_{i=1}^{j-1} l_i\right) l_j^{d-j+1}(w-1)^{j-2}\right) \\
&= \sum_{j=1}^{d}\left(\left(\prod_{i=1}^{j-1} l_i\right) l_j^{d-j+1}(w-1)^{j-1}\right) + \sum_{j=1}^{d-1}\left(\left(\prod_{i=1}^{j-1} l_{i+1}\right) l_{j+1}^{d-j}(l_{j+1} - l_1)(w-1)^{j-1}\right).
\end{aligned}$$

Since $l_{j+1} - l_1 \leq 0$, for $1 \leq j \leq d-1$, the second summation is ≤ 0. Thus

$$\sum_{y \in \mathcal{M}} s_y^d \leq \sum_{j=1}^{d}\left(\left(\prod_{i=1}^{j-1} l_i\right) l_j^{d-j+1}(w-1)^{j-1}\right),$$

which is the desired result. □

Lemma 7.4 *Let S be a set of n data points in U^d. Let T be the partition tree for S built using the balance-split rule. Then the following are true: (i) $\sum_{x \in \mathcal{L}_T} s_x^d = O(\log^{d-1} n)$, and (ii) $\sum_{x \in \mathcal{I}_T} s_x^d = O(\log^d n)$. The constant factors in the O-notation depend on d.*

Proof To prove (i), we apply Lemma 7.3 with x as the root of T, w as the height of T, and $\mathcal{M}$ as the set of leaves $\mathcal{L}_T$. Observe that no leaf can be an ancestor of another leaf, so the conditions of Lemma 7.3 are met. Since the height of T is $O(\log n)$ and $l_i = 1, 1 \leq i \leq d$, we get $\sum_{x \in \mathcal{L}_T} s_x^d \leq \sum_{j=1}^{d} O(\log^{j-1} n)$. Note that this is a geometric series dominated by the last term, which proves (i).

To prove (ii), let $\mathcal{I}_i$ denote the set of internal nodes at level i in T. As in part (i), we can apply Lemma 7.3 to prove $\sum_{x \in \mathcal{I}_i} s_x^d = O(\log^{d-1} n)$. Summing this quantity over all the levels in the tree proves (ii). □

Theorem 7.2 *Let S be a set of n data points in U^d. Let T be the partition tree for S built using the balance-split rule. Given any ϵ, assuming that the query point is chosen from the uniform distribution in U^d, the expected query time of priority search on T is $O((1/\epsilon)^d \log^d n)$. The constant factor in the O-notation depends on d.*

Proof By Lemma 4.2, the query time is $O(I + Ld + L \log I)$, where I and L are the number of internal and leaf nodes visited, respectively. For fixed d, the expected query time is given by $O(E[I] + E[L \log I])$. Since $I \leq n$, the expected query time can be written as $O(E[I] + (\log n)E[L])$. By Lemmas 6.1 and 7.4, $E[I] = O((1/\epsilon)^d \log^d n)$ and $E[L] = O((1/\epsilon)^d \log^{d-1} n)$. Thus the expected query time is $O((1/\epsilon)^d \log^d n)$. □

We mention that it is easy to construct examples that show the tightness of our analysis for the balance-split tree, both in the worst case and in the expected case. Here we are ignoring constant factors depending on d and ϵ.

8 The Sliding-Midpoint Tree

One can easily construct data sets for which the sliding-midpoint tree has $\Omega(n)$ height (see Figure 11) and so its worst-case query time is $\Omega(n)$. Interestingly, even though the tree can be very unbalanced, we will prove that it achieves $O(\log n)$ expected-case query time.

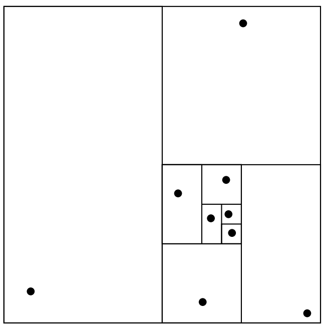

Figure 11: Sliding-midpoint tree: $\Omega(n)$ height.

8.1 Expected-Case Analysis

Recall that the expected query time of priority search on a partition tree T is related to the quantities $\sum_{x \in \mathcal{I}_T} s_x^d$ and $\sum_{x \in \mathcal{L}_T} s_x^d$. The following lemma obtains upper bounds on these quantities for the sliding-midpoint tree.

Lemma 8.1 *Let S be a set of n data points in U^d. Let T be the partition tree for S built using the sliding-midpoint rule. Then the following are true: (i) $\sum_{x \in \mathcal{I}_T} s_x^d = O(\log n)$, and (ii) $\sum_{x \in \mathcal{L}_T} s_x^d = O(1)$. The constant factors in the O-notation depends on d.*

Proof Recall that the sliding-midpoint rule partitions a cell into two subcells by a hyperplane orthogonal to its longest side and its middle. If there are data points in both subcells it does nothing else. Otherwise if there are no data points in one subcell, it slides the splitting plane until it just passes through a data point; the larger child become a leaf and the smaller child becomes an internal node. It follows that the size of an internal node decreases by a factor of at least 2, as we descend d levels in the tree.

Define the level of the root to be 1. For $i \geq 1$, define block i to consist of levels $(i-1)d+1$ to id. Let $\mathcal{I}_i$ denote the set of internal nodes in block i. By the above observation, the size of an internal node in block i is at most $1/2^{i-1}$. Since the number of internal nodes in block i is at most $2^{(i-1)d}(2^d-1)$,

$$\sum_{x \in I_i} s_x^d \leq \left(\frac{1}{2^{i-1}}\right)^d 2^{(i-1)d}(2^d-1) = 2^d - 1.$$

Let $\mathcal{I}'$ be the set of internal nodes in block $i, 1 \leq i \leq \lceil \log n/d \rceil$ and $\mathcal{I}''$ be the remainder of the internal nodes. We will show that $\sum_{x \in \mathcal{I}'} s_x^d = O(\log n)$, and $\sum_{x \in \mathcal{I}''} s_x^d = O(1)$, which will prove (i).

Since $\mathcal{I}'$ consists of $\lceil \log n/d \rceil$ blocks,

$$\sum_{x \in \mathcal{I}'} s_x^d \leq \left\lceil \frac{\log n}{d} \right\rceil (2^d - 1) = O(\log n).$$

It remains to show that $\sum_{x \in \mathcal{I}''} s_x^d = O(1)$. Since each internal node has at least two data points, the number of internal nodes at any level is at most $\lceil n/2 \rceil \leq n$. Thus, the number of internal nodes in a block is $\leq nd$, which implies

$$\sum_{x \in I_i} s_x^d \leq nd \left(\frac{1}{2^{i-1}}\right)^d = \frac{nd}{2^{(i-1)d}}.$$

Therefore,

$$\begin{aligned} \sum_{x \in \mathcal{I}''} s_x^d &\leq \sum_{i > \lceil \frac{\log n}{d} \rceil} \frac{nd}{2^{(i-1)d}} = nd \sum_{i > \lceil \frac{\log n}{d} \rceil} \frac{1}{2^{(i-1)d}} \\ &= nd \left(\frac{1/n}{1 - 1/2^d} \right) \leq 2d = O(1). \end{aligned}$$

This completes the proof of (i).

Next we prove (ii). Recall that for any node x in the tree, R_x is the associated rectangle, defined by the splitting planes used in the construction of the tree. Now we associate a new rectangle R'_x with node x using the following simple procedure. The root is associated with the unit hypercube U^d. Inductively, assume that R'_x is the rectangle associated with node

x. If node x has children in the tree, then split any of the longest sides of R'_x at its middle, and associate the two resulting rectangles with the children of node x.

For each node x, let s'_x denote the longest side of the rectangle R'_x. Since the procedure splits the longest side of the rectangle each time, s'_x decreases by a factor of 2 as we descend d levels in the tree. Thus for any internal node x, $s_x \le s'_x$, and for any leaf node x, $s_x \le 2 \cdot s'_x$, which implies that

$$\sum_{x \in \mathcal{L}_T} s_x^d \le 2^d \sum_{x \in \mathcal{L}_T} (s'_x)^d. \tag{8}$$

Further, since the rectangles R'_x have aspect ratio bounded by 2,

$$\sum_{x \in \mathcal{L}_T} (s'_x)^d \le 2^{d-1} \sum_{x \in \mathcal{L}_T} v'_x, \tag{9}$$

where v'_x denotes the volume of rectangle R'_x. Since the rectangles R'_x associated with the leaves form a subdivision of U^d, $\sum_{x \in \mathcal{L}_T} v'_x = 1$. Combining this with Eq. (8) and (9), we get $\sum_{x \in \mathcal{L}_T} s_x^d \le 2^{2d-1} = O(1)$. □

Using the upper bounds obtained in Lemma 8.1, it is easy to bound the expected query time.

Theorem 8.1 *Let S be a set of n data points in U^d. Let T be the partition tree for S built using the sliding-midpoint rule. Given any ϵ, assuming that the query point is chosen from the uniform distribution in U^d, the expected query time of priority search on T is $O((1/\epsilon)^d \log n)$. The constant factor in the O-notation depends on d.*

Proof By Lemma 4.2, the query time is $O(I + Ld + L \log I)$, where I and L are the number of internal and leaf nodes visited , respectively. For fixed d, the expected query time is given by $O(E[I] + E[L \log I])$. Since $I \le n$, the expected query time can be written as $O(E[I] + (\log n)E[L])$. By Lemmas 6.1 and 8.1, $E[I] = O((1/\epsilon)^d \log n)$ and $E[L] = O((1/\epsilon)^d)$. Thus the expected query time is $O((1/\epsilon)^d \log n)$. □

9 The Hybrid-Split Tree

Recall that the hybrid-split tree is built by alternating between the sliding-midpoint and the balance-split rule at different levels of the tree. We make this more precise. Let t_1 and t_2 be any two fixed constants. We apply the sliding-midpoint rule t_1 times, then apply the balance-split rule t_2 times, and repeat until each cell has one data point. We call this the (t_1, t_2) hybrid-split tree. We get different kinds of trees, depending on the choice of the constants t_1 and t_2.

In the first subsection, we show that the hybrid-split tree yields a worst-case query time of $O((1/\epsilon)^{d(d+1)/2} \log^d n)$. In the second subsection, we show that the hybrid-split tree yields expected-case query time of $O((1/\epsilon)^d \log n \log\log n + (1/\epsilon)^d \log(1/\epsilon) \log n)$.

9.1 Worst-Case Analysis

Since we apply the balance-split rule at least once every constant number of levels, this ensures that the height of the tree is $O(\log n)$. Also, note that both the sliding-midpoint rule and the balance-split rule form child rectangles by splitting the longest side of the parent rectangle. Thus the resulting tree satisfies the conditions of Theorem 7.1, and we have the following corollary.

Corollary 9.1 *Let $t_1, t_2, d > 0$ be any fixed positive integers. Let S be a set of n data points in U^d. Let T be the (t_1, t_2) hybrid-split tree for S. Given any ϵ, the worst-case query time of priority search on T is $O((1/\epsilon)^{d(d+1)/2} \log^d n)$. The constant factor in the O-notation depends on d, t_1, and t_2.*

9.2 Expected-Case Analysis

We give two different analysis of the expected query time on the (t_1, t_2) hybrid-split tree. The first analysis applies for any constants t_1 and t_2. It shows that the expected query time is $O(\log n \log\log n)$. Unfortunately, the constant factor hidden in the O-notation is at least doubly exponential in d. We therefore give a second analysis which, for the special case of the $(2d-2, 1)$ hybrid-split tree, reduces the constant factor to singly exponential in $d \log d$. This analysis is also much simpler than the first.

9.2.1 General Case

We start by defining a *compress* procedure for partition trees, which will facilitate the analysis. Given a binary partition tree T and a positive integer t (called the compression parameter), the new partition tree T' is built as follows. Let $\mathcal{X}$ denote the set of nodes of T at levels $1 + ti, i \geq 0$. Define a bijection f from the nodes of $\mathcal{X}$ to the nodes of T' such that the following is true for any node $x \in \mathcal{X}$. Let the level of x in T be $1 + ti$. Then the level of $f(x)$ in T' is $1 + i$. Further, all the descendants of x at level $1 + t(i+1)$ in T are children of node $f(x)$ in T'. Finally, the rectangle associated with node $f(x)$, denoted $R_{f(x)}$, is the same as R_x. Note that a node of T' can have at most 2^t children.

Let $\mathcal{N}_T$ and $\mathcal{N}_{T'}$ denote the set of nodes of trees T and T', respectively. Then

$$\sum_{x \in \mathcal{N}_T} s_x^d \leq (2^t - 1) \sum_{x \in \mathcal{N}_{T'}} s_x^d. \tag{10}$$

To see this, map each node x of T to the node of T' that corresponds to its nearest ancestor (not necessarily proper) in $\mathcal{X}$. The claim follows since at most $2^t - 1$ nodes of T map to a given node in T', which has larger size.

Recall that the expected query time is related to the quantity $\sum_{x \in \mathcal{N}_T} s_x^d$. In Lemma 9.1 and Corollary 9.2, we establish an upper bound of $O(\log n)$ on this quantity. For any node x of the hybrid-split tree T, let $\mathcal{N}_{x,w}$ denote the set of nodes at levels $\leq w$ in the subtree rooted at x.

Lemma 9.1 *Let $t_1, t_2, d > 0$ be any fixed positive integers. Let S be a set of n data points in U^d. Let T be a (t_1, t_2) hybrid-split tree for S. Let x be any node in T. Then for any w,*

$$\sum_{y \in \mathcal{N}_{x,w}} s_y^d \leq c s_x^d w, \tag{11}$$

where c is a constant depending on d, t_1, and t_2.

Proof Let t_1 and t_2 be fixed as given in the statement of the lemma. We will prove that, for any d, there exists a constant c_d such that Eq. (11) holds with $c = c_d$. The proof is by induction on the dimension d.

Basis: For $d = 1$, observe that the nodes at each level of the subtree rooted at x form a disjoint set of intervals, whose union is a subset of the interval associated with x. Thus $\sum_{y \in \mathcal{N}_{x,w}} s_y^1 \leq s_x w$, and the claim in the lemma holds with $c_1 = 1$.

Induction Step: Assume that the claim in the lemma holds for dimension $\leq d - 1$. We will show that the claim holds for dimension d.

Let x be any node of T. Let T_x denote the subtree rooted at x. Construct tree T'_x from T_x, using the *compress* procedure described earlier. The compression parameter is g_d, where g_d is a suitable constant depending on dimension (we specify it later). To simplify the notation, we write g for g_d in what follows.

For any node u of T'_x and any w, let $\mathcal{N}'_{u,w}$ denote the set of nodes at levels $\leq w$ in the subtree rooted at u. Let $\mathcal{C}'_u$ denote the set of children of u in T'_x. By the same reasoning as in Eq. (10),

$$\sum_{y \in \mathcal{N}_{x,w}} s_y^d \leq (2^g - 1) \left(\sum_{y \in \mathcal{N}'_{f(x), \lceil w/g \rceil}} s_y^d \right). \tag{12}$$

Here f is the mapping defined in the *compress* procedure.

We have the following claim. Let z be any node of T'_x. Then for any w,

$$\sum_{y \in \mathcal{N}'_{z,w}} s_y^d \leq 2 s_z^d + 4 \cdot 2^g (1 + 2^{g+1}) v_z (w - 1). \tag{13}$$

The proof is by induction on w. For the basis, observe that when $w = 1$, the left hand side of the inequality is s_z^d and the right hand side is $2 s_z^d$.

Assume next that the claim holds for $w < k$. We will prove the claim for $w = k$. Let z be any node of T'_x. Clearly,

$$\sum_{y \in \mathcal{N}'_{z,k}} s_y^d = s_z^d + \sum_{u \in \mathcal{C}'_z} \left(\sum_{y \in \mathcal{N}'_{u,k-1}} s_y^d \right).$$

By the induction hypothesis,

$$\begin{aligned} \sum_{y \in \mathcal{N}'_{z,k}} s_y^d &\leq s_z^d + \sum_{u \in \mathcal{C}'_z} \left(2 s_u^d + 4 \cdot 2^g (1 + 2^{g+1}) v_u (k - 2) \right) \\ &= s_z^d + 2 \left(\sum_{u \in \mathcal{C}'_z} s_u^d \right) + 4 \cdot 2^g (1 + 2^{g+1}) v_z (k - 2), \end{aligned} \tag{14}$$

where we have used the fact that $\sum_{u \in \mathcal{C}'_z} v_u = v_z$. We consider two cases.

Case 1: R_z has no side of length $\leq s_z/(4^{1/d}2^{g/d})$.

Rewriting Eq. (14),

$$\sum_{y \in \mathcal{N}'_{z,k}} s_y^d \leq s_z^d + 2\sum_{u \in \mathcal{C}'_z} s_u^d - 4 \cdot 2^g(1+2^{g+1})v_z + 4 \cdot 2^g(1+2^{g+1})v_z(k-1).$$

Since z has at most 2^g children each of size $\leq s_z$ and $v_z \geq s_z^d/(4 \cdot 2^g)$,

$$\begin{aligned}\sum_{y \in \mathcal{N}'_{z,k}} s_y^d &\leq s_z^d + 2(2^g)s_z^d - 4 \cdot 2^g(1+2^{g+1})\left(\frac{s_z^d}{4 \cdot 2^g}\right) + 4 \cdot 2^g(1+2^{g+1})v_z(k-1) \\ &\leq 2s_z^d + 4 \cdot 2^g(1+2^{g+1})v_z(k-1).\end{aligned}$$

Case 2: R_z has at least one side of length $\leq s_z/(4^{1/d}2^{g/d})$.

We will prove that

$$\sum_{u \in \mathcal{C}'_z} s_u^d \leq \frac{s_z^d}{2}. \tag{15}$$

Assuming this for now and substituting it in Eq. (14),

$$\begin{aligned}\sum_{y \in \mathcal{N}'_{z,k}} s_y^d &\leq s_z^d + 2\left(\frac{s_z^d}{2}\right) + 4 \cdot 2^g(1+2^{g+1})v_z(k-2) \\ &\leq 2s_z^d + 4 \cdot 2^g(1+2^{g+1})v_z(k-1).\end{aligned}$$

This completes the proof by induction of Eq. (13).

It remains to prove Eq. (15). Let p denote the node of T that corresponds to node z (i.e., $p = f^{-1}(z)$). Note that $R_p = R_z$. Let $\mathcal{M}$ denote the set of nodes at level $g+1$ in the subtree rooted at p. From the definition of T'_z, there is a 1-1 correspondence between the nodes in $\mathcal{C}'_z$ and the nodes in $\mathcal{M}$, and corresponding nodes are associated with the same rectangle. Thus

$$\sum_{u \in \mathcal{C}'_z} s_u^d = \sum_{u \in \mathcal{M}} s_u^d.$$

We partition the set $\mathcal{M}$ into two disjoint sets $\mathcal{M}_1$ and $\mathcal{M}_2$: the nodes in $\mathcal{M}$ that have size $\leq s_z/(4^{1/d}2^{g/d})$ belong to $\mathcal{M}_1$, and the rest belong to $\mathcal{M}_2$. We show that this sum is at most $s_z^d/4$ for each set, which will establish Eq. (15).

By definition of $\mathcal{M}_1$,

$$\sum_{u \in \mathcal{M}_1} s_u^d \leq 2^g\left(\frac{s_z}{4^{1/d}2^{g/d}}\right)^d \leq \frac{s_z^d}{4}.$$

Next we show that $\sum_{u \in \mathcal{M}_2} s_u^d \leq s_z^d/4$. Let b denote the side of R_z with length $\leq s_z/(4^{1/d}2^{g/d})$ (there must exist such a side). Let u be any node in $\mathcal{M}_2$. Consider the path from p to u in the tree T. Note that R_u is formed from $R_z(= R_p)$ by applying g splits to R_z. Clearly, none of these splits divides side b of the rectangle, because otherwise s_u would be $\leq s_z/(4^{1/d}2^{g/d})$ and node u would be assigned to $\mathcal{M}_1$.

Let $e = \lceil d/t_1 \rceil t_2 + d + 1$, and $g = e \lceil f \log(2c_{d-1}e) \rceil$, where f is a suitable constant > 1 independent of dimension (to be specified later).

Recall that the sliding-midpoint rule decreases the length of the longest side of the rectangle by a factor of at least 2 for any resulting internal nodes. It is easy to see that any set of $e - 1$ consecutive splits will contain d or more applications of the sliding-midpoint rule. Thus, the size of internal nodes after $e - 1$ splits, and the size of any node after e splits, will decrease by a factor of at least 2. Since any node $u \in \mathcal{M}_2$ is formed from R_z by applying g splits, it follows that $s_u \leq s_z/(2c_{d-1}e)^f$. Thus,

$$\sum_{u\in\mathcal{M}_2} s_u^d \leq \sum_{u\in\mathcal{M}_2} \left(s_u^{d-1} \cdot \frac{s_z}{(2c_{d-1}e)^f} \right) \leq \frac{s_z}{(2c_{d-1}e)^f} \sum_{u\in\mathcal{M}_2} s_u^{d-1}. \tag{16}$$

Recall that the nodes of $\mathcal{M}_2$ are formed from p without ever splitting side b; that is, the g splits divide the remaining $d-1$ sides of the rectangle. We can therefore use the induction hypothesis (Eq. (11)) to bound $\sum_{u\in\mathcal{M}_2} s_u^{d-1}$. We obtain

$$\sum_{u\in\mathcal{M}_2} s_u^{d-1} \leq c_{d-1} s_z^{d-1}(1 + g).$$

Recalling $g = e \lceil f \log(2c_{d-1}e) \rceil$, and using the above bound in Eq. (16), we obtain

$$\begin{aligned}
\sum_{u\in\mathcal{M}_2} s_u^d &\leq \frac{s_z}{(2c_{d-1}e)^f} c_{d-1} s_z^{d-1}(1 + e \lceil f \log(2c_{d-1}e) \rceil) \\
&= s_z^d \left(\frac{c_{d-1}(1 + e \lceil f \log(2c_{d-1}e) \rceil)}{(2c_{d-1}e)^f} \right).
\end{aligned}$$

It is easy to see that for suitable f (independent of d), the expression in big parenthesis can be made as small as we like, in particular, $\leq 1/4$. This completes the proof of Eq. (15).

Finally, combining Eq. (12) and (13),

$$\sum_{y\in\mathcal{N}_{x,w}} s_y^d \leq (2^g - 1) \left(2s_x^d + 4 \cdot 2^g (1 + 2^{g+1}) v_x \left(\left\lceil \frac{w}{g} \right\rceil - 1 \right) \right).$$

Since $v_x \leq s_x^d$, this yields

$$\sum_{y\in\mathcal{N}_{x,w}} s_y^d \leq c_d s_x^d w,$$

where $c_d = (2^g - 1)(2 + 4 \cdot 2^g(1 + 2^{g+1}))/g$, which completes the proof of the lemma. □

We apply Lemma 9.1 with x set to the root of the hybrid-split tree, and w set to its height, which is $O(\log n)$. Since the rectangle associated with the root of the tree is the unit hypercube, it follows that $s_x = 1$, and we obtain the following corollary.

Corollary 9.2 *Let $t_1, t_2, d > 0$ be any fixed positive integers. Let S be a set of n data points in U^d. Let T be a (t_1, t_2) hybrid-split tree for S. Let $\mathcal{I}_T$ and $\mathcal{L}_T$ denote the set of internal nodes and leaves in T. Then the following are true: (i) $\sum_{x \in \mathcal{L}_T} s_x^d = O(\log n)$, and (ii) $\sum_{x \in \mathcal{I}_T} s_x^d = O(\log n)$. The constant factors in the O-notation depend on d, t_1, and t_2.*

Theorem 9.1 *Let $t_1, t_2, d > 0$ be any fixed positive integers. Let S be a set of n data points in U^d. Let T be a (t_1, t_2) hybrid-split tree for S. Given any ϵ, assuming that the query point is chosen from the uniform distribution in U^d, the expected query time of priority search on T is $O((1/\epsilon)^d \log n \log\log n + (1/\epsilon)^d \log(1/\epsilon) \log n)$. The constant factor in the O-notation depends on d, t_1, and t_2.*

Proof By Lemma 4.2, the query time is $O(I + Ld + L \log I)$, where I and L are the number of internal and leaf nodes visited, respectively. For fixed d, the expected query time is $O(E[I] + E[L \log I])$. From Corollary 9.1, the number of internal nodes seen in the worst case is $O((1/\epsilon)^{d(d+1)/2} \log^d n)$. Thus the expected query time is $O(E[I]) + O((\log\log n + \log(1/\epsilon))E[L])$.

By Lemma 6.1 and Corollary 9.2, $E[I] = O((1/\epsilon)^d \log n)$ and $E[L] = O((1/\epsilon)^d \log n)$, which implies that the expected query time is $O((1/\epsilon)^d \log n \log\log n + (1/\epsilon)^d \log(1/\epsilon) \log n)$.
□

9.2.2 Special Case

Examining the proof of Lemma 9.1, it is easy to see that we get a bound on the constant c in the statement of the lemma that is at least doubly exponential in d. Therefore, the constant factor in the expected query time, derived in Theorem 9.1, is also at least doubly exponential in d. We now give a simple analysis of the $(2d-2, 1)$ hybrid-split tree, which gives a bound on the constant factor that is singly exponential in $d \log d$.

Let T denote the $(2d-2, 1)$ hybrid-split tree. We start by computing the quantity $\sum_{x \in \mathcal{N}_T} s_x^d$, which is related to the expected query time by Lemma 6.1. To this end, define a new tree T', which is obtained from T by the *compress* procedure defined in the last subsection. The compression parameter is $2d-1$. Using Eq. (10), we have

$$\sum_{y \in \mathcal{N}_T} s_y^d \leq (2^{2d-1} - 1) \sum_{y \in \mathcal{N}_{T'}} s_y^d. \tag{17}$$

In Lemma 9.3, we compute an upper bound on $\sum_{x \in \mathcal{N}_{T'}} s_x^d$. The lemma relies on the following observation.

Lemma 9.2 *Let x be any node of T'. If R_x has at least one side of length $\leq s_x/4$, then the size of any child of x is at most $s_x/4$.*

Proof Let y be any child of x. Let x' and y' be the nodes in T corresponding to nodes x and y. That is, $x' = f^{-1}(x)$ and $y' = f^{-1}(y)$. Consider the path from x' to y' in the tree T. Recall that $R_x = R_{x'}$ and $R_y = R_{y'}$. Note that R_y is formed from R_x by applying $2d-2$ sliding-midpoint splits followed by one balance-split. Let z denote the parent of y'. We will show that the size of R_z is at most $s_x/4$, from which the lemma will follow.

Since z is an internal node, it is clear that the longest side of the rectangle decreases by a factor of at least 2 at each of the $2d-2$ steps of the path from x' to z. Let k denote the side of R_x that has length $\leq s_x/4$. We consider three cases.

First, assume R_z is formed from R_x by splitting side k. Since the longest side is split each time, it implies that all the sides of R_z have length $\leq s_x/4$.

Second, assume that R_z is formed from R_x by splitting some side of R_x more than twice. Note that when this side is split a third time, its length must be $\leq s_x/4$. Since the side split is the longest side, all sides of R_z have length $\leq s_x/4$.

Otherwise, assume that neither of these two cases holds. That is, R_z is formed from R_x without splitting side k and without splitting any side of R_x more than twice. Since $2d-2$ splits are applied, it follows that each of the $d-1$ sides other than side k is split exactly twice. It follows that these $d-1$ sides each have length $\leq s_x/4$. Since side k also has length $\leq s_x/4$, this completes the proof. $\square$

For any node u of T', let $\mathcal{N}'_{u,w}$ denote the set of nodes at levels $\leq w$ in the subtree rooted at u, and let $\mathcal{C}'_u$ denote the set of children of u in T'.

Lemma 9.3 *Let x be any node of T'. Then for any w,*

$$\sum_{y \in \mathcal{N}'_{x,w}} s_y^d \leq 2s_x^d + 2^{2d}(1+2^{2d})v_x(w-1).$$

Proof The proof is by induction on w. For the basis, observe that when $w=1$, the left hand side of the inequality is s_x^d and the right hand side is $2s_x^d$.

Assume next that the claim in the lemma holds for $w<k$. We will prove the claim for $w=k$. Let x be any node of T'. Clearly,

$$\sum_{y \in \mathcal{N}'_{x,k}} s_y^d = s_x^d + \sum_{z \in \mathcal{C}'_x} \left(\sum_{y \in \mathcal{N}'_{z,k-1}} s_y^d \right).$$

By the induction hypothesis,

$$\begin{aligned} \sum_{y \in \mathcal{N}'_{x,k}} s_y^d &\leq s_x^d + \sum_{z \in \mathcal{C}'_x} \left(2s_z^d + 2^{2d}(1+2^{2d})v_z(k-2) \right) \\ &= s_x^d + 2\left(\sum_{z \in \mathcal{C}'_x} s_z^d \right) + 2^{2d}(1+2^{2d})v_x(k-2), \end{aligned} \tag{18}$$

where we have used the fact that $\sum_{z \in \mathcal{C}'_x} v_z = v_x$. We consider two cases.

Case 1: R_x has no side of length $\leq s_x/4$.

Rewriting Eq. (18),

$$\sum_{y \in \mathcal{N}'_{x,k}} s_y^d \leq s_x^d + 2\sum_{z \in \mathcal{C}'_x} s_z^d - 2^{2d}(1+2^{2d})v_x + 2^{2d}(1+2^{2d})v_x(k-1).$$

Since x has at most 2^{2d-1} children each of size $\leq s_x$ and $v_x \geq (s_x/4)^d$,

$$\begin{aligned}\sum_{y \in \mathcal{N}'_{x,k}} s_y^d &\leq s_x^d + 2(2^{2d-1})s_x^d - 2^{2d}(1+2^{2d})\left(\frac{s_x}{4}\right)^d + 2^{2d}(1+2^{2d})v_x(k-1) \\ &\leq 2s_x^d + 2^{2d}(1+2^{2d})v_x(k-1).\end{aligned}$$

Case 2: R_x has at least one side of length $\leq s_x/4$.
By Lemma 9.2, all nodes $z \in \mathcal{C}'_x$ satisfy $s_z \leq s_x/4$. Substituting in Eq. (18), we get

$$\begin{aligned}\sum_{y \in \mathcal{N}'_{x,k}} s_y^d &\leq s_x^d + 2(2^{2d-1})\left(\frac{s_x}{4}\right)^d + 2^{2d}(1+2^{2d})v_x(k-2) \\ &\leq 2s_x^d + 2^{2d}(1+2^{2d})v_x(k-1).\end{aligned}$$

This completes the proof by induction. □

We apply Lemma 9.3 with x set to the root of T' and w set to its height, which is $O(d \log n)$. Since the rectangle associated with the root of T' is the unit hypercube, it follows that $s_x = 1$ and $v_x = 1$. Thus we get $\sum_{y \in \mathcal{N}_{T'}} s_y^d = 2^{O(d)} \log n$. Combining with Eq. (17), we get $\sum_{y \in \mathcal{N}_T} s_y^d = 2^{O(d)} \log n$. The following theorem can now be proved in the same way as Theorem 9.1.

Theorem 9.2 *Let S be a set of n data points in U^d. Let T be the $(2d-2, 1)$ hybrid-split tree for S. Given any ϵ, assuming that the query point is chosen from the uniform distribution in U^d, the expected query time of priority search on T is $O((1/\epsilon)^d \log n \log\log n + (1/\epsilon)^d \log(1/\epsilon)\log n)$. The constant factor in the O-notation is $2^{O(d \log d)}$.*

10 Optimality of the Sliding-Midpoint Tree

In Section 8, we proved that the expected query time of priority search on the sliding-midpoint tree is $O(\log n)$. If the data points are uniformly distributed, then $\Omega(\log n)$ is a lower bound on the expected query time in the decision tree model. Thus, the sliding-midpoint tree achieves optimal expected query time for uniformly distributed data points (ignoring constant factors depending on d and ϵ). The question naturally arises whether the sliding-midpoint tree achieves minimum expected query time for other data distributions as well. We prove that this is indeed the case under reasonable assumptions.

Let S be a set of n data points in U^d. Let $\mathcal{T}_S$ denote the class of partition trees for S formed by splits using arbitrary oriented hyperplanes (i.e., *binary space partition trees*) such that each leaf cell has at most one data point. We have the following result:

Theorem 10.1 *Let d and ϵ be any fixed constants. There exists a constant c depending on d and ϵ such that the following is true. Let S be any set of n data points in U^d, and let T be any partition tree in $\mathcal{T}_S$. Assuming that the query point is chosen from the uniform distribution in U^d, the expected query time of priority search on the sliding-midpoint tree is no more than c times the expected query time of priority search on T.*

We begin with some definitions. Given a region z, let v_z denote its volume (area, in 2D, and length, in 1D). Let $\mathcal{Z}$ denote any set of regions inside the unit hypercube U^d (note that the regions need not form a subdivision of U^d, nor are they necessarily disjoint). We define the *entropy* of $\mathcal{Z}$ to be $\sum_{z \in \mathcal{Z}} v_z \log(1/v_z)$. The entropy of a subdivision of the unit hypercube is defined to be the entropy of the set of regions that form the subdivision. In this section, to avoid confusion, we always use *data point* to refer to a point in the data set S, and use *point* to refer to any point in space.

We give a brief overview of the proof. Let S be a set of n data points in U^d. Let $\mathcal{D}$ denote any set of cells in U^d that satisfy the following properties, for some constants c_a and c_n, depending on dimension:

(A.1) *Difference of Two Rectangles:* A cell is the set-theoretic difference of two axis-parallel rectangles, one enclosed within the other. We call these the *outer rectangle* and *inner rectangle* of the cell. Note that the inner rectangle need not be present.

(A.2) *Bounded Aspect Ratio:* The outer rectangle and inner rectangle (if present) have aspect ratio (ratio of longest to shortest side) $\leq c_a$. (In this case we say that the cell has aspect ratio $\leq c_a$.)

(A.3) *Stickiness:* If the cell has an inner rectangle, then for each dimension, the separation between the corresponding faces of the inner and outer rectangle is either 0 or at least the length of the inner rectangle along that dimension.

(A.4) *Existence of a Close Data Point:* Define the size of a cell to be the length of the longest side of its outer rectangle. There is a data point whose distance from any point inside the outer rectangle of the cell is $\leq c_n s$, where s is the size of the cell.

(A.5) *Disjointness:* Given any two cells in $\mathcal{D}$, either the outer rectangles of the two cells are disjoint or the outer rectangle of one cell is contained within the inner rectangle of the other.

The basic idea of the proof is to show that the entropy of any such set of cells is a lower bound on the expected query time of priority search on any partition tree. For the upper bound, we will determine a set $\mathcal{D}$ of cells in U^d satisfying properties A.1–A.5, such that the expected query time of priority search on the sliding-midpoint tree is no more than the entropy of $\mathcal{D}$ (ignoring constant additive and multiplicative factors depending on d and ϵ).

10.1 Lower Bound

The main result of this section is the following:

Lemma 10.1 *Let d and ϵ be any fixed constants. Let S be any set of n data points in U^d, and let T be any partition tree in $\mathcal{T}_S$. Let $\mathcal{D}$ be any set of cells in U^d satisfying properties A.1–A.5. Assuming that the query point is chosen from the uniform distribution in U^d, the expected query time of priority search on T is $\Omega(entropy(\mathcal{D}) + 1)$.*

Briefly, the proof works as follows. Given any partition tree $T \in \mathcal{T}_S$, let $\mathcal{Z}_T$ denote the subdivision of the unit hypercube induced by the leaf nodes of T. Since each internal node splits the associated region into two parts by a hyperplane, it follows that the cells in $\mathcal{Z}_T$ are convex polytopes. Further, each cell in $\mathcal{Z}_T$ is either empty or contains one data point. Let $\mathcal{Z}'_T$ denote the subdivision formed by splitting each non-empty cell in $\mathcal{Z}_T$ into two parts by passing a hyperplane through the data point inside it. The cells in $\mathcal{Z}'_T$ are convex polytopes in U^d and satisfy the following properties, for constant c_v, depending on dimension (property B.1 is obvious; property B.2 is proved in Lemma 10.2):

(B.1) *Empty Interior:* A cell contains no data point in its interior.

(B.2) *Proportionality of Swept Volume to Radius:* Let z denote the cell, $\ominus$ denote the Minkowski difference operator, and B_r denote a ball of radius r. For any $r \geq 0$, the volume of $z - (z \ominus B_r)$ (i.e., the set of points inside z that are within distance r of the boundary of z) is at most $c_v r$.

Let $\mathcal{D}$ be any set of cells in U^d satisfying properties A.1–A.5 and let $\mathcal{Z}$ be any subdivision of U^d into cells satisfying properties B.1–B.2. Lemmas 10.3, 10.4, and 10.5 form the cornerstone of the lower bound argument, and show that the entropy of $\mathcal{D}$ is $O(\mathit{entropy}(\mathcal{Z}) + 1)$. Since $\mathcal{Z}'_T$ satisfies properties B.1–B.2, it follows that the entropy of $\mathcal{D}$ is $O(\mathit{entropy}(\mathcal{Z}'_T)+1)$. It is easy to show that $\mathit{entropy}(\mathcal{Z}'_T) \leq \mathit{entropy}(\mathcal{Z}_T)+1$. Thus $\mathit{entropy}(\mathcal{D}) = O(\mathit{entropy}(\mathcal{Z}_T)+1)$, as shown in Lemma 10.6. Finally, Lemma 10.7 proves that the expected query time of priority search on T is $\Omega(\mathit{entropy}(\mathcal{Z}_T) + 1)$. Together, this implies that the expected query time of priority search on T is $\Omega(\mathit{entropy}(\mathcal{D}) + 1)$.

We start by proving that any convex polytope in U^d satisfies property B.2 (for constant $c_v = 2d$).

Lemma 10.2 *Let z be a convex polytope contained within U^d. For any $r \geq 0$, the volume of $z - (z \ominus B_r)$ is at most $2dr$.*

Proof Let $y = z - (z \ominus B_r)$. By definition, y is the set of points inside z at distance at most r from the boundary of z. For $d = 1$, z is a line segment and it is obvious that the volume of y is at most $2r$.

Let $d > 1$. For each facet of z, construct a prism with this facet as base, directed inwards into z, and with height r. It is easy to see that any point in y must lie in one of these prisms. Thus the volume of y is no more than the volume of all the prisms, which is clearly $a_z r$, where a_z is the surface area (perimeter, in 2D) of z. Since $z \subseteq U^d$, a_z is no more than the surface area of U^d, which is $2d$. (Here we have employed the following fact: if A and B are two closed, bounded, and convex subsets of R^d and $A \subseteq B$, then the surface area of A is no more than the surface area of B. See [14] for a proof.) Thus, the volume of y is at most $2dr$. $\square$

Now we are ready to present the key lemma for the lower bound argument. In view of the applications of the lemma to other problems (such as planar point location [1]), we prove a stronger version than is strictly needed here. Given a cell u satisfying property A.1 (difference of two rectangles), let u_O and u_I denote its outer and inner rectangle, respectively.

Lemma 10.3 *Let d be any fixed constant. Let $\mathcal{Z}$ be any set of disjoint cells in U^d satisfying property B.2 (for constant c_v), and let $\mathcal{D}$ be any set of cells in U^d satisfying properties A.1, A.2 (for constant c_a), A.3, and A.5. Assume further that there exists a constant c_n (depending on dimension) such that, for any cell $u \in \mathcal{D}$ and $z \in \mathcal{Z}$, if $u_O \subseteq z$, then the distance of any point in u_O from the boundary of z is at most $c_n s_u$. Define a* fragment *to be a connected component in the intersection of a cell in $\mathcal{Z}$ with a cell in $\mathcal{D}$. Let $\mathcal{F}$ be the set of all fragments. Then*

$$entropy(\mathcal{F}) \leq d \cdot entropy(\mathcal{Z}) + O\left(\sum_{z \in \mathcal{Z}} v_z\right),$$

where the constant in the O-notation depends on d.

Proof For any cell $z \in \mathcal{Z}$, let $\mathcal{F}_z \subseteq \mathcal{F}$ denote the set of fragments that are contained within z. We claim that $entropy(\mathcal{F}_z) \leq dv_z \log(1/v_z) + O(v_z)$. Since $entropy(\mathcal{F}) = \sum_{z \in \mathcal{Z}} entropy(\mathcal{F}_z)$, the lemma follows by summing over all $z \in \mathcal{Z}$.

Let z be any cell in $\mathcal{Z}$. In the remainder, we will prove that $entropy(\mathcal{F}_z) \leq dv_z \log(1/v_z) + O(v_z)$. We start by partitioning z into an infinite set of regions, denoted $z_i, i \geq 1$, as follows. Let $r_c = v_z/c_v$. Define z'_i to be the region consisting of points inside z at distance at least $r_c/2^i$ from the boundary of z. Clearly $z'_i \subseteq z'_{i+1}$ for $i \geq 1$. By property B.2 and the definition of r_c, it follows that the volume of z'_i is at least $v_z(1 - 1/2^i)$. The volume of z'_1 is thus at least $v_z/2$; we define z_1 to be any region of volume $v_z/2$ inside z'_1. Next observe that the volume of z'_2 is at least $3v_z/4$ and so the volume of $z'_2 - z_1$ is at least $v_z/4$; we define z_2 to be any region of volume $v_z/4$ inside z'_2 that is disjoint from z_1. Continuing in this way, we define $z_i, i \geq 1$, to be any region of volume $v_z/2^i$ inside z'_i that is disjoint from all of the regions $z_1, z_2, \ldots, z_{i-1}$. It is clear that the regions $z_i, i \geq 1$ form an infinite partition of z and satisfy the following two conditions: $v_{z_i} = v_z/2^i$ and $z_i \subseteq z'_i$.

Define a *sub-fragment* to be the intersection of a fragment in $\mathcal{F}_z$ with a region $z_i, i \geq 1$. Let $\mathcal{G}_z$ denote the set of all sub-fragments. Since each fragment $y \in \mathcal{F}_z$ is partitioned into sub-fragments $y \cap z_i, i \geq 1$, it is easy to see that

$$entropy(\mathcal{F}_z) \leq entropy(\mathcal{G}_z). \tag{19}$$

Let $\mathcal{G}_{z_i}$ denote the set of sub-fragments that lie inside z_i. Clearly

$$entropy(\mathcal{G}_z) = \sum_{i \geq 1} entropy(\mathcal{G}_{z_i}). \tag{20}$$

Next we compute an upper bound on $entropy(\mathcal{G}_{z_i})$; the desired bound on $entropy(\mathcal{F}_z)$ will follow using Eqs. (19) and (20). Obviously $entropy(\mathcal{G}_{z_i}) \leq entropy(\mathcal{G}_{z_i} \cup \{x\})$, where $x = z_i - \bigcup \mathcal{G}_{z_i}$ (i.e., the region remaining in z_i after removing all the sub-fragments in $\mathcal{G}_{z_i}$). Let m_i be the number of sub-fragments in $\mathcal{G}_{z_i}$. It follows from basic calculus that $entropy(\mathcal{G}_{z_i} \cup \{x\})$ can be no more than the entropy of the set of regions formed by spliting z_i into $m_i + 1$ parts of equal volume. Thus

$$entropy(\mathcal{G}_{z_i}) \leq v_{z_i} \log \frac{m_i + 1}{v_{z_i}} \tag{21}$$

We will show that $m_i \le c \cdot 2^{id}/v_z^{d-1}$, where c is a constant that depends on dimension d. Assuming this fact for now and recalling that $v_{z_i} = v_z/2^i$ we get

$$entropy(\mathcal{G}_{z_i}) \le \frac{v_z}{2^i} \log \frac{c2^{id}/v_z^{d-1} + 1}{v_z/2^i} \le \frac{v_z}{2^i} \log \frac{(c+1)2^{id}/v_z^{d-1}}{v_z/2^i}.$$

Letting $c' = c + 1$ and simplifying we get

$$entropy(\mathcal{G}_{z_i}) \le \left(dv_z \log \frac{1}{v_z} + v_z \log c' \right) \frac{1}{2^i} + (d+1)v_z \frac{i}{2^i}.$$

Using Eq. (20) we obtain

$$entropy(\mathcal{G}_z) \le dv_z \log \frac{1}{v_z} + v_z(\log c' + 2(d+1)) = dv_z \log \frac{1}{v_z} + O(v_z),$$

where the constant in the O-notation depends on d. By Eq. (19), $entropy(\mathcal{F}_z) \le dv_z \log(1/v_z) + O(v_z)$, which is the desired claim.

It remains to show that $m_i \le c \cdot 2^{id}/v_z^{d-1}$. Recall that m_i is the number of fragments that overlap z_i. Since $z_i \subseteq z'_i$, it follows that m_i is no more than the number of fragments that overlap z'_i. Let $\mathcal{F}_{z'_i}$ denote the set of fragments that overlap z'_i. For convenience set $r_i = r_c/2^i$ and $c'_n = max(c_n, d)$. Below we give a method to associate a distinct hypercube of size $r_i/(c'_n c_a)$ with at least a fraction $1/(2d)$ of the fragments in $\mathcal{F}_{z'_i}$. The resulting set of hypercubes $\mathcal{H}$ are disjoint and contained within z. A simple packing argument then implies that

$$|\mathcal{H}| \le \frac{v_z}{(r_i/(c'_n c_a))^d}.$$

Substituting $r_i = r_c/2^i$ and $r_c = v_z/c_v$, we get

$$|\mathcal{H}| \le \frac{(c'_n c_a c_v)^d 2^{id}}{v_z^{d-1}}.$$

The number of fragments in $\mathcal{F}_{z'_i}$ is at most $2d$ times this quantity. We thus obtain the desired bound $m_i \le c \cdot 2^{id}/v_z^{d-1}$, where $c = 2d(c'_n c_a c_v)^d$.

Let $u \in \mathcal{D}$ be any cell that contains a fragment of $\mathcal{F}_{z'_i}$. Clearly u overlaps z'_i. We claim that the size of u is at least r_i/c'_n. If u_O overlaps the boundary of z, then the diameter of u_O must exceed r_i and so s_u is at least r_i/d. Otherwise u_O is contained within z and, by the statement of the lemma, the distance of any point in u_O from the boundary of z is at most $c_n s_u$. Since u_O overlaps z'_i, it contains a point at distance at least r_i from the boundary of z; thus $c_n s_u \ge r_i$ which implies that $s_u \ge r_i/c_n$. Combining the two cases we get $s_u \ge r_i/c'_n$, where $c'_n = max(c_n, d)$.

We now describe how to associate hypercubes with fragments in $\mathcal{F}_{z'_i}$; later we will show that the resulting set of hypercubes satisfy all the properties mentioned earlier. We use H_y to denote the hypercube (if any) associated with fragment y. For each fragment $y \in \mathcal{F}_{z'_i}$ choose a point p_y in $y \cap z'_i$ (we call such a point a *fragment point*). Let $u \in \mathcal{D}$ be any cell that overlaps z'_i. We consider three cases: (i) u has no inner rectangle, (ii) u has an inner rectangle, which has size $\geq r_i/c'_n$, and (iii) u has an inner rectangle, which has size $< r_i/c'_n$. If u has no inner rectangle, then for each fragment $y \subseteq u$, choose a hypercube H_y of size $r_i/(c'_n c_a)$ that overlaps p_y and is contained within u. Note there exists such a hypercube since $s_u \geq r_i/c'_n$ and by property A.2 (bounded aspect ratio) the length of each side of u is at least $r_i/(c'_n c_a)$.

In the second case, u has an inner rectangle of size $\geq r_i/c'_n$. Observe that u can be partitioned into t rectangles, where $1 \leq t \leq 2d$ is the number of sides of the inner rectangle that do not touch the corresponding side of the outer rectangle. (This can be done by successively passing hyperplanes that touch a side of the inner rectangle. For example, Figure 12(a), (b), and (c) show how a cell in two dimensions is partitioned into 2, 3, and 4 rectangles, respectively.) Note that by properties A.2 (bounded aspect ratio) and A.3 (stickiness), each side of these t rectangles is of length at least $r_i/(c'_n c_a)$. Of these t rectangles, let R denote the rectangle that has the maximum number of fragment points. For each fragment y such that $p_y \in R$, choose a hypercube H_y of size $r_i/(c'_n c_a)$ that overlaps p_y and is contained within R. For the remaining fragments in u (i.e., fragments whose corresponding fragment point lies in $u - R$), we associate no hypercube.

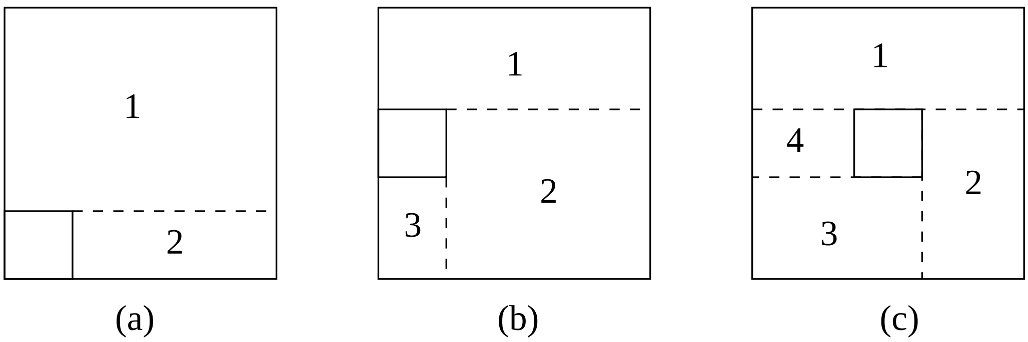

Figure 12: Partitioning a cell into rectangles.

In the third case, u has an inner rectangle of size $< r_i/c'_n$. As in the second case, we partition u into t rectangles, $1 \leq t \leq 2d$, and determine the rectangle R among them that has the maximum number of fragment points. For each fragment y such that $p_y \in R$, we choose a hypercube H_y of size $r_i/(c'_n c_a)$ that overlaps p_y and is contained within the outer rectangle of u. For the remaining fragments in u, we associate no hypercube. (Note that, unlike the second case, the sides of R may be of length $< r_i/(c'_n c_a)$ and the hypercube H_y may overlap the inner rectangle of u.)

It remains to argue that the resulting set of hypercubes $\mathcal{H}$ has the desired properties. Let y denote a fragment in $\mathcal{F}_{z'_i}$. By construction, the size of H_y (if defined) is $r_i/(c'_n c_a)$. Second, since H_y overlaps p_y and p_y is at distance at least r_i from the boundary of z, it follows that H_y is contained within z. Third, for any cell u that overlaps z'_i, it is clear that we associate a hypercube with at least a fraction $1/(2d)$ of the fragments of $\mathcal{F}_{z'_i}$ contained in u. It follows that a hypercube is associated with at least a fraction $1/(2d)$ of the fragments in $\mathcal{F}_{z'_i}$.

The only thing left to show is that the hypercubes are all disjoint. Let $y_1, y_2 \in \mathcal{F}_{z'_i}$ be any two distinct fragments that each have a hypercube associated with them. Let u_1

and u_2 be the cells containing y_1 and y_2, respectively. There are four different cases: (i) $u_{1O} \cap u_{2O} = \emptyset$, (ii) $u_{2O} \subseteq u_{1I}$, (iii) $u_{1O} \subseteq u_{2I}$, and (iv) $u_1 = u_2$ (i.e., y_1 and y_2 are contained in the same cell).

By our construction $H_{y_1} \subseteq u_{1O}$ and $H_{y_2} \subseteq u_{2O}$. It follows that in Case (i), H_{y_1} and H_{y_2} must be disjoint. In Case (ii), recall that a cell containing a fragment of $\mathcal{F}_{z_i'}$ must have size at least r_i/c_n'. Thus u_2 has size at least r_i/c_n', and hence u_{1I} has size at least r_i/c_n'. By our construction, this implies that $H_{y_1} \subseteq u_1$. Since $H_{y_2} \subseteq u_{2O}$ it follows that H_{y_1} and H_{y_2} are disjoint. Case (iii) is similar to the second case. In Case (iv), observe that the line segment joining p_{y_1} and p_{y_2} lies entirely within cell u_1 (this is obvious if u_1 has no inner rectangle; if u_1 has an inner rectangle, this follows from the fact that we partition u_1 into rectangles and associate hypercubes with fragment points lying in only one of these rectangles). Since y_1 and y_2 are distinct fragments, the boundary of z must intersect this line segment. Since the distance of both p_{y_1} and p_{y_2} from the boundary of z is at least r_i, the distance between p_{y_1} and p_{y_2} must be at least $2r_i$. Since H_{y_1} and H_{y_2} overlap p_{y_1} and p_{y_2}, respectively, and each has size $r_i/(c_n' c_a)$, it is easy to verify that H_{y_1} and H_{y_2} are disjoint. This completes the proof of the lemma. $\square$

Lemma 10.4 *Let d be any fixed constant. Let $\mathcal{Z}$ and $\mathcal{D}$ be sets of cells satisfying all the conditions given in the statement of Lemma 10.3. Further, let $\mathcal{F}$ be as defined in the statement of Lemma 10.3. Assuming that $\mathcal{Z}$ is a subdivision of U^d,*

$$\mathit{entropy}(\mathcal{D}) \leq \mathit{entropy}(\mathcal{F}) \leq d \cdot \mathit{entropy}(\mathcal{Z}) + O(1),$$

where the constant in the O-notation depends on d.

Proof Clearly, the set of fragments in $\mathcal{F}$ form a refinement of the set of cells in $\mathcal{D}$. Thus, $\mathit{entropy}(\mathcal{D}) \leq \mathit{entropy}(\mathcal{F})$. Using Lemma 10.3 and the fact that $\sum_{z \in \mathcal{Z}} v_z = 1$, we get $\mathit{entropy}(\mathcal{F}) \leq d \cdot \mathit{entropy}(\mathcal{Z}) + O(1)$, which completes the proof. $\square$

Lemma 10.5 *Let d be any fixed constant. Let S be any set of n data points in U^d. Let $\mathcal{Z}$ be a subdivision of U^d whose cells satisfy properties B.1–B.2, and let $\mathcal{D}$ be a set of cells in U^d satisfying properties A.1–A.5. Then*

$$\mathit{entropy}(\mathcal{D}) = O(\mathit{entropy}(\mathcal{Z}) + 1),$$

where the constant in the O-notation depends on d.

Proof Let u be a cell in $\mathcal{D}$ whose outer rectangle is contained within some cell $z \in \mathcal{Z}$. We claim that its distance from the boundary of z is at most $c_n s_u$. This follows from the fact that the distance between any point in the outer rectangle of u and the nearest data point is at most $c_n s_u$ (property A.4) and z contains no data point in its interior (property B.1). Thus, $\mathcal{D}$ and $\mathcal{Z}$ satisfy the conditions of Lemma 10.4, which implies the desired claim. $\square$

Lemma 10.6 *Let d be any fixed constant. Let S be any set of n data points in U^d, and let T be any partition tree in $\mathcal{T}_S$. Let $\mathcal{Z}_T$ be the subdivision of U^d induced by the leaf nodes of T. Then $entropy(\mathcal{D}) = O(entropy(\mathcal{Z}_T)+1)$, where $\mathcal{D}$ is any set of cells in U^d satisfying properties A.1–A.5. The constant in the O-notation depends on d.*

Proof Refine the subdivision $\mathcal{Z}_T$ by splitting each non-empty cell of $\mathcal{Z}_T$ into two parts by passing any hyperplane through the data point inside the cell. Let $\mathcal{Z}'_T$ be the new subdivision. It is easy to see that $entropy(\mathcal{Z}'_T)$ can be no more than the entropy of the set of regions formed by splitting each cell of $\mathcal{Z}_T$ into two parts of equal volume. Thus,

$$entropy(\mathcal{Z}'_T) \leq \sum_{z \in \mathcal{Z}_T} 2\frac{v_z}{2} \log \frac{1}{v_z/2} = \sum_{z \in \mathcal{Z}_T} \left(v_z \log \frac{1}{v_z} + v_z \right) = entropy(\mathcal{Z}_T) + 1, \tag{22}$$

where we have used the fact that $\sum_{z \in \mathcal{Z}_T} v_z = 1$.

Clearly, $\mathcal{Z}'_T$ is a subdivision of U^d into cells satisfying property B.1 (empty interior). Also, since the cells in $\mathcal{Z}'_T$ are convex polytopes in U^d, by Lemma 10.2, they satisfy property B.2 (proportionality of swept volume to radius). Thus, $\mathcal{D}$ and $\mathcal{Z}'_T$ satisfy the conditions of Lemma 10.5, which implies that

$$entropy(\mathcal{D}) = O(entropy(\mathcal{Z}'_T) + 1). \tag{23}$$

The lemma now follows from Eqs. (22) and (23). □

Lemma 10.7 *Let d and ϵ be any fixed constants. Let S be any set of n data points in U^d, and let T be any partition tree in $\mathcal{T}_S$. Let $\mathcal{Z}_T$ be the subdivision of U^d induced by the leaf nodes of T. Assuming that the query point is chosen from the uniform distribution in U^d, the expected query time of priority search on T is $\Omega(entropy(\mathcal{Z}_T)+1)$.*

Proof Given a node z in the partition tree T, let d_z denote its depth. Recall that the algorithm starts by descending from the root of the tree to the leaf that contains the query point. Thus, the expected number of nodes visited by the algorithm to locate the leaf cell containing the query point is $\sum_{z \in \mathcal{Z}_T}(v_z \cdot d_z)$. Note that this is the weighted path length of the tree, where the weight of a leaf is the volume of the associated cell. A fundamental informtation theoretic result due to Shannon [25] implies that the weighted path length of any binary tree with these weights is at least $\sum_{z \in \mathcal{Z}_T} v_z \log(1/v_z)$. Thus $entropy(\mathcal{Z}_T)$ is a lower bound on the expected number of nodes visited. Noting that the algorithm must visit at least one node, the lemma follows. □

Finally, Lemmas 10.6 and 10.7 together imply Lemma 10.1.

10.2 Upper Bound

In this subsection, we compute an upper bound on the expected query time of priority search on the sliding-midpoint tree. We will prove that the expected query time is $O(entropy(\mathcal{D})+1)$, where $\mathcal{D}$ is some set of cells in U^d satisfying properties A.1–A.5 listed in the last subsection. In view of Lemma 10.1, this would imply the optimality of the sliding-midpoint tree in the sense of Theorem 10.1.

In order to gain intuition, we first analyze the expected query time of priority search on a partition tree called the *box-decomposition (BD) tree* [4, 8]. We give a simple analysis which shows that this is $O(entropy(\mathcal{D})+1)$, where $\mathcal{D}$ is some set of cells in U^d satisfying properties A.1–A.5. The cells of the BD-tree can have holes, which makes it less practical than the sliding-midpoint tree. Therefore, we later extend the analysis to the sliding-midpoint tree, and show that a similar upper bound holds.

10.2.1 Expected Query Time Using the BD-tree

First we mention some properties of the BD-tree that follow direcly from its construction [4, 8]. Let T denote the BD-tree and let $\mathcal{Z}_T$ denote the subdivision of U^d induced by the leaf nodes of T. The cells in $\mathcal{Z}_T$ satisfy properties A.1–A.5. Also, the cells associated with the internal nodes of T satisfy properties A.1–A.4 given earlier. In addition, the BD-tree has the following property, which is important for our analysis:

(A.6) *Geometric Decrease in Volume:* The volume of the cells associated with the nodes decreases by at least a constant factor every $O(1)$ levels of descent in the tree. This property implies that if the volume of a leaf is v, then its depth is $O(\log(1/v)+1)$.

In Lemma 10.9 we prove that the expected query time of priority search on the BD-tree is $O(\sum_{x\in\mathcal{N}_T} s_x^d)$, and in Lemma 10.10 we prove that $\sum_{x\in\mathcal{N}_T} s_x^d = O(entropy(\mathcal{Z}_T)+1)$. Setting $\mathcal{D} = \mathcal{Z}_T$, it follows that the expected query time is $O(entropy(\mathcal{D})+1)$, where $\mathcal{D}$ is some set of cells in U^d satisfying properties A.1–A.5.

We need the following packing lemma proved by Arya et al. [4, 5].

Lemma 10.8 *The number of leaf cells of the BD-tree visited by priority search in the worst case is* $O((1+1/\epsilon)^d)$.

This result holds irrespective of the data distribution and the location of the query point. Intuitively, this lemma holds because the leaf cells satisfy the bounded aspect ratio and stickiness properties.

Lemma 10.9 *The expected query time of priority search on the BD-tree is* $O(\sum_{x\in\mathcal{N}_T} s_x^d)$.

Proof By Lemma 4.2, the query time is $O(I+Ld+L\log I)$, where I and L are the number of internal and leaf nodes visited, respectively. Thus, for fixed d and ϵ, the expected query time is $O(E[I]+E[L\log I])$. By Lemma 10.8, L is bounded by a constant. Hence, the expected query time is $O(E[I]+E[\log I]) = O(E[I])$. By Lemma 6.1, $E[I] = O(\sum_{x\in\mathcal{I}_T} s_x^d)$, which implies the desired result. □

In the next lemma, we show that there exists a set of cells in U^d satisfying properties A.1–A.4, whose entropy is an upper bound on the quantity $O(\sum_{x\in\mathcal{N}_T} s_x^d)$.

Lemma 10.10 *Let T be any tree in which cells associated with the leaf and internal nodes of the tree satisfy properties A.1–A.3 and A.6. Let $\mathcal{Z}_T$ denote the subdivision of U^d induced by the leaf nodes of T. Then* $\sum_{x\in\mathcal{N}_T} s_x^d = O(entropy(\mathcal{Z}_T)+1)$.

Proof Since the cells associated with the nodes of the tree satisfy the bounded aspect ratio and stickiness properties, it follows that $\sum_{x \in \mathcal{N}_T} s_x^d = O\left(\sum_{x \in \mathcal{N}_T} v_x\right)$. Since the volume of a node x is the sum of the volume of all the leaf nodes descended from it, we can write $\sum_{x \in \mathcal{N}_T} v_x = \sum_{x \in \mathcal{L}_T} (v_x \cdot d_x)$, where d_x denotes the depth of leaf x. Further, by property A.6, $d_x = O(\log(1/v_x) + 1)$. Thus $\sum_{x \in \mathcal{N}_T} s_x^d = O\left(\sum_{x \in \mathcal{L}_T} v_x (\log(1/v_x) + 1)\right) = O(\mathit{entropy}(\mathcal{Z}_T) + 1)$. □

Recalling the features of the BD-tree mentioned earlier, and applying Lemmas 10.9 and 10.10, we obtain the following upper bound on the expected query time.

Lemma 10.11 *The expected query time of priority search on the BD-tree is $O(\mathit{entropy}(\mathcal{D}) + 1)$, where $\mathcal{D}$ is some set of cells in U^d satisfying properties A.1–A.5.*

Finally, combining Lemmas 10.1 and 10.11, establishes the optimality of the BD-tree in the sense described in Theorem 10.1.

10.2.2 Expected Query Time Using the Sliding-Midpoint Tree

The main problem in establishing a similar upper bound for the sliding-midpoint tree is that the rectangles associated with the nodes of the tree can be skinny. In particular, this implies that neither Lemma 10.1 nor Lemma 10.10 can be directly applied. We solve this problem by showing that, for any sliding-midpoint tree T, there exists a closely related tree T' having essentially the same features as the BD-tree, and such that any node of T maps to a distinct node of T' of similar size. We prove this in Lemma 10.12. The rest of the analysis is similar to that for the BD-tree.

Lemma 10.12 *Let S be a set of n data points in U^d. Let T be the sliding-midpoint tree for S. Then there exists a partition tree T' such that*

(1) The cells associated with the nodes (internal and leaf) of tree T' satisfy properties A.1–A.4 and A.6. Also, the set of cells associated with the leaves of T' satisfy property A.5.

(2) There exists a 1-1 mapping f from the nodes of T to the nodes of T' such that for any node x of T, $s_x/2 \leq s_{f(x)} \leq 8s_x$.

Proof

We will construct the partition tree T', and the mapping f, incrementally in $n-1$ steps. To this end, we label the internal nodes of T from 1 to $n-1$, such that the label assigned to any child is greater than the label assigned to its parent. Clearly, there exists such a labeling (for example, label the internal nodes in a breadth-first manner). Let T_i denote the subtree of T consisting of the root and the nodes whose parents have label $\leq i$.

Let T'_i denote the tree constructed after i steps. In addition to this tree, we also maintain a 1-1 mapping f from the nodes of T_i to the nodes of T'_i. We will prove by induction that the following invariant holds at each step of the construction. (For convenience we denote the cell associated with a node x by R_x if it is a rectangle and C_x if it is the difference of two rectangles.)

(1) T'_i is a partition tree, and f is a 1-1 mapping from the nodes of T_i to the nodes of T'_i.

(2) Properties A.1–A.4 and A.6 hold for the cells associated with the nodes (internal and leaf) in T'_i.

(3) Property A.5 holds for the set of cells associated with the leaves of T'_i.

(4) Let x be any internal node in T that is present in T_i. Let $y = f(x)$. Then

   (a) The cell associated with node y is a rectangle (denoted R_y) and $R_x \subseteq R_y$,

   (b) each side of R_y has length $\geq s_x/2$ and $\leq 4s_x$, and

   (c) if x is a leaf in T_i, then y is a leaf in T'_i.

(5) Let x be any leaf in T that is present in T_i. Let $y = f(x)$. Then $s_x/2 \leq s_y \leq 8s_x$.

It is clear that the lemma will then follow from the fact that the invariant holds for the final tree $T' = T'_{n-1}$.

Initially, T'_0 consists of just the root node, which is associated with U^d. Further, the root of T is mapped by f to the root of T'_0. It is easy to see that the invariant holds for $i = 0$. We now describe the ith step of the construction. Let x denote the internal node of T labelled i. Observe that our method of labelling implies that x is a leaf in T_{i-1}. Thus, by (4c) of invariant, the corresponding node $y = f(x)$ must be a leaf in T'_{i-1}. Let x_1 and x_2 denote the two children of x. In the ith step, we *attach* a subtree consisting of a constant number of nodes to node y, and map x_1 and x_2 by function f to two leaf nodes in this subtree. For the other nodes in T_i, the mapping f remains unchanged (note they are all present in T_{i-1}).

Now we give the details. Let the longest side of R_x that is split to form R_{x_1} and R_{x_2} be aligned along the kth coordinate axis, and let P denote the hyperplane orthogonal to it and passing through the center of R_x. We consider two cases.

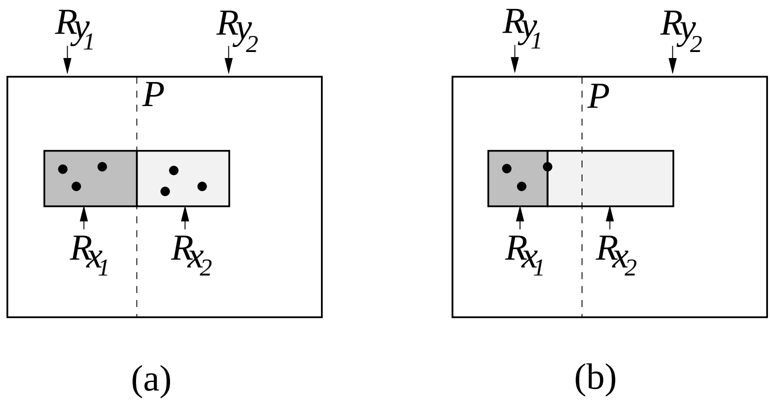

Figure 13: Proof of Lemma 10.12.

Case 1: P is the splitting hyperplane associated with node x. (See Figure 13(a).)

This implies that P splits R_x into R_{x_1} and R_{x_2}, and there is a data point in each of these two rectangles. We modify T'_{i-1} in two sub-steps, each sub-step adds at most one level to T'_{i-1}, and we show that the invariant holds after the second sub-step.

Sub-step 1: Create two child nodes y_1 and y_2 for node y. By (4a) of invariant, the cell associated with y is a rectangle R_y. and $R_x \subseteq R_y$. We describe how to partition R_y into two disjoint cells, which are associated with y_1 and y_2. We split R_y into two rectangles by hyperplane P, and associate these rectangles with y_1 and y_2 such that R_{y_1} and R_{x_1} are both on the same side of P, and R_{y_2} and R_{x_2} are both on the other side of P. Henceforth, in our discussion of Case 1 we focus only on nodes x_1 and y_1, since nodes x_2 and y_2, respectively, play a symmetrical role.

We show that properties A.1–A.4 hold for R_{y_1}. First, observe that since R_{y_1} is a rectangle with no hole, it trivially satisfies property A.1 (difference of two rectangles) and property A.3 (stickiness). Second, since $R_x \subseteq R_y$, it follows that $R_{x_1} \subseteq R_{y_1}$. Since there is a data point in R_{x_1}, it implies that there is a data point in R_{y_1} as well, and so R_{y_1} satisfies property A.4 (existence of a close data point).

Third, we claim that each side of R_{y_1} has length $\geq s_x/2$ and $\leq 4s_x$. Note that this would imply that R_{y_1} satisfies property A.2 (bounded aspect ratio). From (4b) of invariant, each side of R_y has length $\geq s_x/2$ and $\leq 4s_x$. Since R_{y_1} is formed from R_y by splitting side k (i.e., side aligned along the kth coordinate axis), the claim is obviously true for all sides of R_{y_1} other than side k. Further, the length of side k of R_{y_1} must be $\leq 4s_x$, since it must be smaller than corresponding side of R_y. It remains to show that the length of side k of R_{y_1} is $\geq s_x/2$. Recall that $R_{x_1} \subseteq R_{y_1}$. Also, side k of R_{x_1} has length $s_x/2$, since P splits R_x at the middle of side k, which is one of the longest sides of R_x. Thus, the length of side k of R_{y_1} is $\geq s_x/2$.

Next we claim that the volume of R_{y_1} is at most a constant fraction of the volume of R_y. As we will see, sub-step 2 adds at most one level to T'_{i-1}; thus, volume of the nodes decreases by a constant fraction after every two levels of descent in T'_i, which implies that T'_i satisfies property A.6 at the end of sub-step 2. To see the claim, observe that the ratio of the volume of R_{y_2} to the volume of R_y is the same as the ratio of the length of side k of R_{y_2} to the length of side k of R_y. Since the length of any side of R_{y_2} is $\geq s_x/2$, and the length of any side of R_y is $\leq 4s_x$, this ratio is at least 1/8. Thus, the volume of R_{y_1} is at most 7/8th the volume of R_y.

Sub-step 2: We expand each side of R_{x_1} that is smaller than $s_{x_1}/2$ to $s_{x_1}/2$. Let B_1 denote this new rectangle. Note that it is possible to do the expansion such that B_1 lies inside R_{y_1}, since $R_{x_1} \subseteq R_{y_1}$ and each side of R_{y_1} is of length $\geq s_x/2 \geq s_{x_1}/2$. Further, each side of B_1 has length $\geq s_{x_1}/2$ and $\leq s_{x_1}$.

Next, if rectangle B_1 has a side that violates the stickiness property with respect to the corresponding side of R_{y_1}, expand it until it touches the side of R_{y_1}. Continue this process, until all sides of the resulting rectangle (call it B_2) satisfy the stickiness property. It is easy to see that this can increase the length of a side of B_1 by a factor of at most 4. Thus, the length of each side of B_2 is $\geq s_{x_1}/2$ and $\leq 4s_{x_1}$.

If $R_{y_1} = B_2$, then set $f(x_1) = y_1$. Otherwise, create two child nodes y_{11} and y_{12} for node y_1, associate rectangle B_2 with y_{11} and $R_{y_1} - B_2$ with y_{12}, and set $f(x_1) = y_{11}$. Note that y_{12} will be a leaf in the final tree T'.

We now show that the invariant holds after the second sub-step. We have already seen that T'_i satisfies property A.6 and properties A.1–A.4 hold for the node y_1 added in sub-step 1. We now show that properties A.1–A.4 hold for the children (if any) added to y_1.

If $f(x_1) = y_1$, then no new nodes are added and there is nothing to vshow. Otherwise nodes y_{11} and y_{12} are made children of y_1. Recall that $R_{y_{11}} = B_2$ and $C_{y_{12}} = R_{y_1} - B_2$, and the length of each side of B_2 is $\geq s_{x_1}/2$ and $\leq 4s_{x_1}$. It follows that $R_{y_{11}}$ and $C_{y_{12}}$ satisfy property A.1 (difference of two rectangles) and property A.2 (bounded aspect ratio). Since B_2 has no hole, it satisfies property A.3 (stickiness). By construction of B_2, it is clear that $C_{y_{12}}$ also satisfies stickiness. Since $R_{x_1} \subseteq B_2$, B_2 has a data point inside it. Since $B_2 \subseteq R_{y_1}$, R_{y_1} also has a data point inside it. Thus B_2 and $C_{y_{12}}$ both satisfy property A.4 (existence of a close data point).

Next we show that (4a–c) of the invariant hold if x_1 is an internal node of T, and (5) of the invariant holds if x_1 is a leaf node. To prove (4a), note that by construction $R_{f(x_1)} = B_2$, a rectangle with no hole, and $R_{x_1} \subseteq B_2$ (since B_2 is formed by expanding R_{x_1}). Recall that each side of B_2 is of length $\geq s_{x_1}/2$ and $\leq 4s_{x_1}$, which implies (4b) and (5). Since $f(x_1)$ is a leaf in the tree T'_i, it follows that (4c) holds.

Finally it is clear from our construction that T'_i is a partition tree whose leaves satisfy property A.4, and f is a 1-1 mapping from the nodes of T_i to the nodes of T'_i ((1) and (3) of invariant).

Case 2: P is not the splitting hyperplane associated with node x. (See Figure 13(b).)

This implies that all the data points in R_x are on the same side of P, and the splitting hyperplane for x is obtained by sliding P along dimension k until it just passes through a data point. Without loss of generality, suppose that R_{x_1} is smaller than R_{x_2}, as shown in Figure 13(b) (the other case can be handled similarly). Note that x_2 is a leaf containing exactly one data point.

In sub-step 1 we create two child nodes y_1 and y_2 for node y. As in Case 1, the cell associated with node y is a rectangle R_y and $R_x \subseteq R_y$. We split R_y into two rectangles by hyperplane P; the rectangle which is on the same side of P as R_{x_1} is associated with y_1 and the other rectangle is associated with y_2. Next we set $f(x_2) = y_2$. Note that y_2 will be a leaf in the final tree T'.

The description of sub-step 2 is identical to Case 1 and so we omit it. Note that we only need to process node x_1 in sub-step 2 (since x_2 is already mapped by f in sub-step 1.)

Now we show that the invariant holds. The argument is the same as for Case 1, with two differences: one, in showing that there is a data point close to R_{y_2} (property A.4), and two, in showing that the size of R_{y_2} is $\geq s_{x_2}/2$ and $\leq 8s_{x_2}$ ((5) of invariant).

Since R_x contains a data point, and $R_x \subseteq R_y$, the distance between any point in R_y and a data point is no more than the diameter of R_y; by (4b) of invariant, this is $\leq 4ds_x$. Since $R_{y_2} \subseteq R_y$, this bound also applies to the distance between any point in R_{y_2} and a data point. Further, the argument given in Case 1 shows that the length of each side of R_{y_2} is $\geq s_x/2$ and $\leq 4s_x$. It follows that R_{y_2} satisfies property A.4.

To show (5) for leaf node x_2, note that R_{x_2} is the larger of the two children formed by applying the sliding-midpoint rule to R_x. Thus, $s_x \geq s_{x_2} \geq s_x/2$. Thus, the length of each side of R_{y_2} is $\geq s_{x_2}/2$ and $\leq 8s_{x_2}$.

□

The following lemma proved in [23] gives a bound on the number of leaf cells visited by priority search.

Lemma 10.13 *([23]) The number of leaf cells of the sliding-midpoint tree visited by priority search in the worst case is bounded by* $O((1 + 1/\epsilon)^d)$.

Let T denote the sliding-midpoint tree, and let T' denote the tree corresponding to it, given in the statement of Lemma 10.12.

Lemma 10.14 *The expected query time of priority search on the sliding-midpoint tree is* $O(\sum_{x \in \mathcal{N}_{T'}} s_x^d)$.

Proof Using Lemma 10.13 and proceeding as in Lemma 10.9, the expected query time is $O(\sum_{x \in \mathcal{N}_T} s_x^d)$. By Lemma 10.12, there is a 1-1 function f that maps each node in T to a node of similar size in T', which proves the lemma. □

Applying Lemma 10.10 to tree T', we obtain $\sum_{x \in \mathcal{N}_{T'}} s_x^d = O(entropy(\mathcal{Z}_{T'}) + 1)$. Setting $\mathcal{D} = \mathcal{Z}_{T'}$, we have the following result similar to Lemma 10.11.

Lemma 10.15 *The expected query time of priority search on the sliding-midpoint tree is* $O(entropy(\mathcal{D}) + 1)$, *where* $\mathcal{D}$ *is some set of cells in* U^d *satisfying properties A.1–A.5.*

Combining this lemma with the lower bound given in Lemma 10.1, establishes Theorem 10.1.

Remark: It is easy to see that our proof of Theorem 10.1 also implies the following: Let S be any set of n data points in U^d, and let A be any algorithm for finding the approximate nearest neighbor that can be embodied as a decision tree T using degree-one tests. Then the expected query time of priority search on the sliding-midpoint tree is no more than a constant (depending on d and ϵ) times the expected query time of A.

We only mention two key observations. First, the leaf nodes of T induce a subdivision $\mathcal{Z}_T$ of U^d into convex cells, such that each cell has at most one data point (since the same data point must be the approximate nearest neighbor, no matter where the query point lies in the cell). Second, the weighted path length of T, where the volume of a leaf is its weight, is a lower bound on the expected query time of A. Thus, as in Section 10.1, we can prove that the expected query time of A is $\Omega(entropy(\mathcal{D}) + 1)$, where $\mathcal{D}$ is any set of cells in U^d satisfying properties A.1–A.5. The claim now follows in light of Lemma 10.15.

11 Experimental Results

We ran experiments to compare the performance of five different partition trees based on the sliding midpoint rule, the balance-split rule, the $(d, 1)$ hybrid-split rule, the standard kd-tree split rule, and the midpoint-split rule. We implemented our algorithms in C++, and tested them using `ANN` [24].

We start by listing the point distributions used for generating the data sets [5, 24]. The clustered segments distribution was used to model data sets which exhibit clustering in low dimensional subspaces. The correlated Gaussian and correlated Laplacian point distributions were chosen to model data from speech processing applications. These two distributions were formed by grouping the output of autoregressive sources into vectors of length d. An autoregressive source uses the following recurrence to generate successive outputs:

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

where W_n is a sequence of zero mean independent, identically distributed random variables. The correlation coefficient ρ was taken as 0.9 for our experiments. Each point was generated by selecting its first coordinate from the corresponding uncorrelated distribution (either Gaussian or Laplacian), and then the remaining coordinates were generated by the equation above. See Farvardin and Modestino [15] for more information.

The two clustered distributions were designed to model data sets where clustering is present. In the clustered Gaussian distribution, points are clustered around a small number of randomly chosen cluster center points. In the clustered segments distribution, the points are clustered around a small number of randomly chosen orthogonal line segments.

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

Clustered Segments: Eight axis-parallel line segments were sampled from a hypercube as follows. For each line segment a random coordinate axis x_k was selected, and a point p was sampled uniformly from the hypercube. The line segment is the intersection of the hypercube with the line parallel to x_k, passing through p. An equal number of points were generated uniformly along the length of each line segment and a Gaussian error with standard deviation of 0.001 was added.

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

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

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

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

We generated data points in dimension 16 from these distributions, and in each case generated query points uniformly from a hypercube enclosing 90% of the data points (to reduce the effect of outliers). Throughout we used a bucket size (the maximum number of data points inside a leaf cell) of one for the partition trees.

For each experiment, we fixed ϵ and measured a number of statistics, averaged over 200 query points. The statistics included the average number of nodes visited, and the average number of floating point operations (i.e., any arithmetic operation involving point coordinates or distances). Both these measures provide reasonable machine-independent measures of the query time.

The results for the uniform, clustered segments, correlated Gaussian, and correlated Laplacian distributions are shown in Figures 14 through 17. Figures 14 and 15 show the average number of floating point operations and nodes visited, respectively, as a function

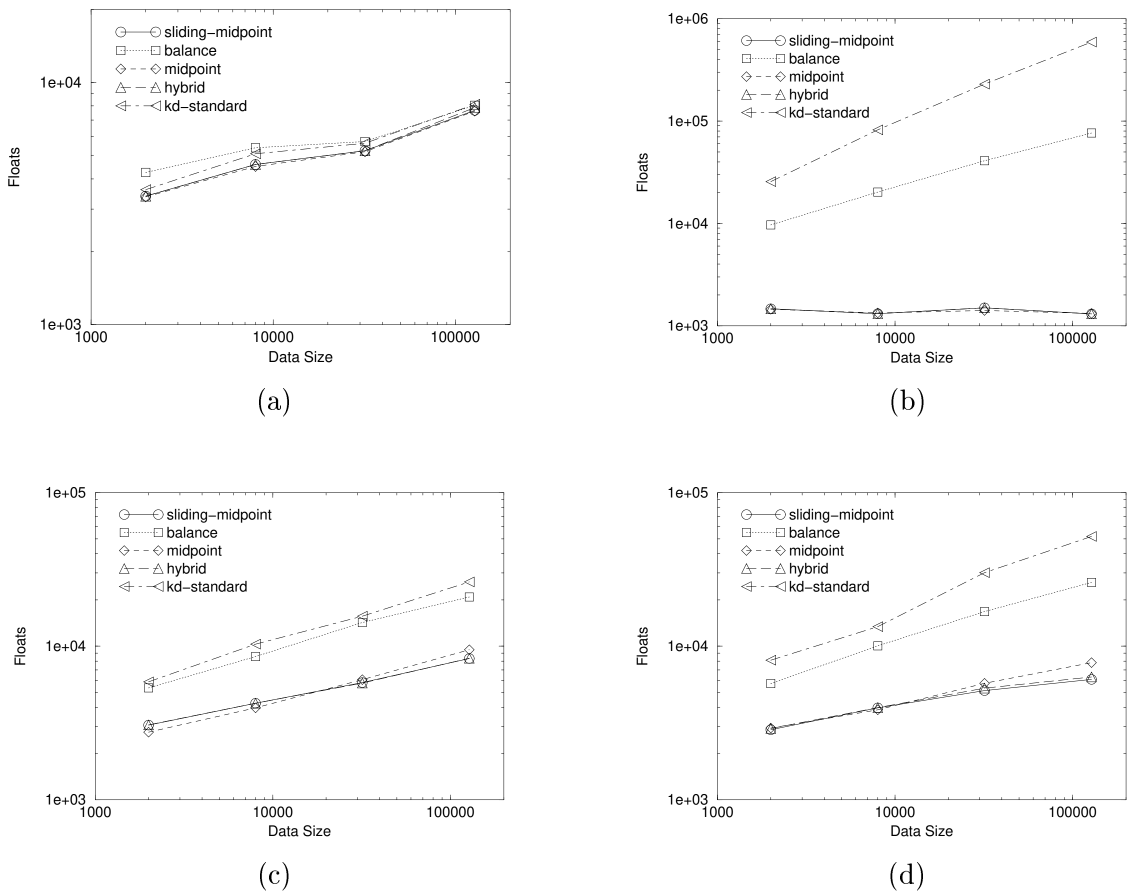

Figure 14: Average number of floating point operations for the (a) uniform, (b) clustered segments, (c) correlated Gaussian, and (d) correlated Laplacian distributions versus n. Here $\epsilon = 2$.

of n, the number of data points. Here ϵ is fixed at 2. Figures 16 and 17 show the average number of floating point operations and nodes, respectively, as a function of ϵ. Here $n = 128,000$.

We can make the following observations from the plots.

- The key observation is that for the clustered and the two correlated distributions, the sliding-midpoint tree and the hybrid-split tree offer significant speed-up, sometimes by factors of over 10, compared to the standard kd-tree. Moreover, the speed-up increases with n.
- For the uniform distribution, all trees yield very similar query times.
- In all cases tested, the balance-split tree performs worse than the sliding-midpoint and hybrid-split trees, but better than the standard kd-tree.
- Even though the midpoint-split tree is worse than the sliding-midpoint tree in terms of the number of nodes visited, it is quite similar in terms of floating point operations.

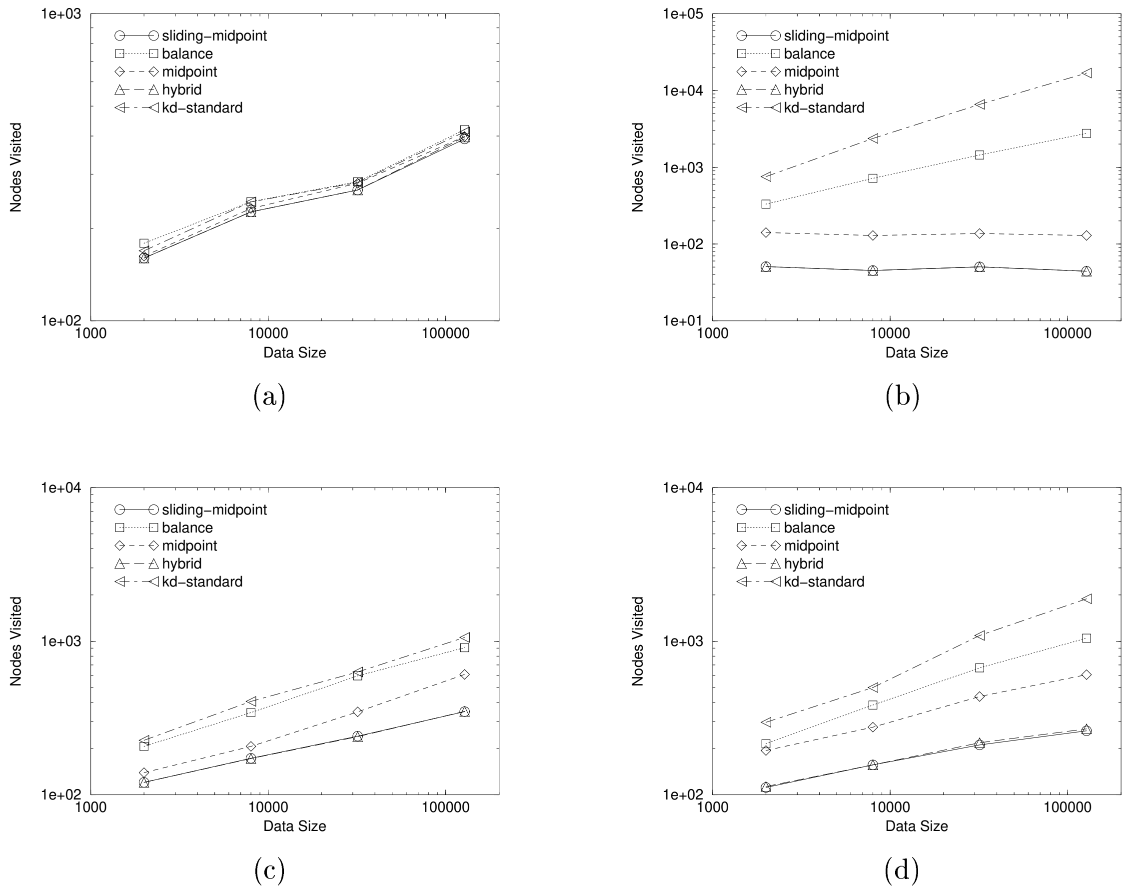

Figure 15: Average number of nodes visited for the (a) uniform, (b) clustered segments, (c) correlated Gaussian, and (d) correlated Laplacian distributions versus n. Here $\epsilon = 2$.

This is because the midpoint-split rule creates a large number of empty leaves which, however, have negligible overhead in terms of floating point operations. (But since they do increase the space requirements, we feel there is a clear reason for preferring the sliding-midpoint tree to the midpoint-split tree.)

- For the clustered segments distribution, the expected number of nodes visited for the midpoint-split, sliding-midpoint, and hybrid-split trees appears to be independent of the number of data points. (This is not hard to show using Lemma 6.1.)
- As ϵ increases from 0 to 1, the query times of all the trees decreases significantly, usually by factors between 10 and 100.

12 Conclusion

We have designed and analyzed algorithms for approximate nearest neighbor searching from the perspective of their expected-case performance. The expected-case analysis makes no

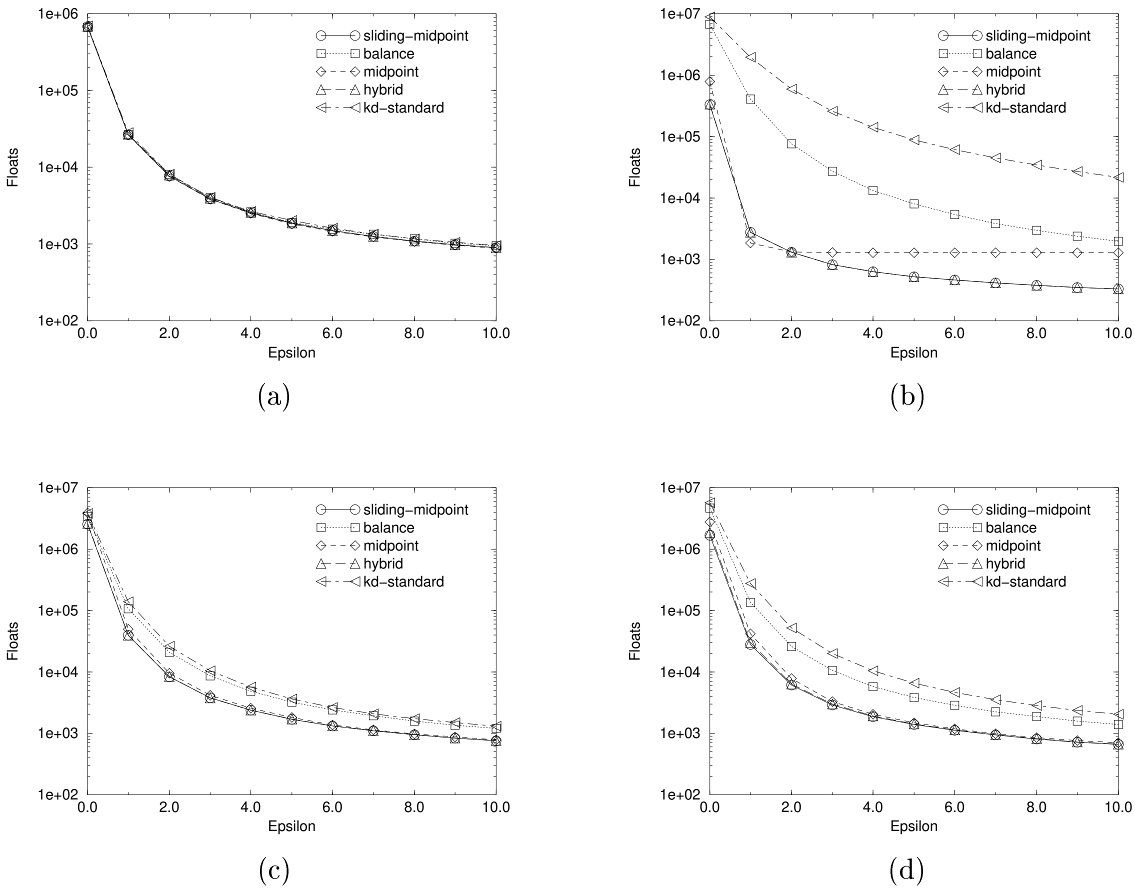

Figure 16: Average number of floating point operations for the (a) uniform, (b) clustered segments, (c) correlated Gaussian, and (d) correlated Laplacian distributions versus ϵ. Here $n = 128,000$.

assumption on the distribution of data points, but assumes that the query points are chosen uniformly from a hypercube enclosing all the data points.

We showed that with very simple partition trees, it is possible to achieve linear space and logarithmic or polylogarithmic expected query time. We also showed that the sliding-midpoint tree attains optimal expected query time (ignoring constant factors) under reasonable assumptions. Compared to data structures used to achieve linear space and logarithmic query time in the worst case, our data structures are simpler and easier to implement.

There are several interesting open problems. The main limitation of our work is that it is restricted to the case of uniform query distribution. It would be interesting to develop an algorithm that achieves optimal expected query time for non-uniform query distributions. We believe that the techniques developed in this paper may prove useful for that task.

Another interesting problem concerns strengthening the optimality claim for the sliding-midpoint tree. We proved that the sliding-midpoint tree has a smaller expected query time (ignoring constant factors) than any algorithm that can be embodied as a decision tree using linear tests. Does this optimality claim hold even if we allow polynomials of higher

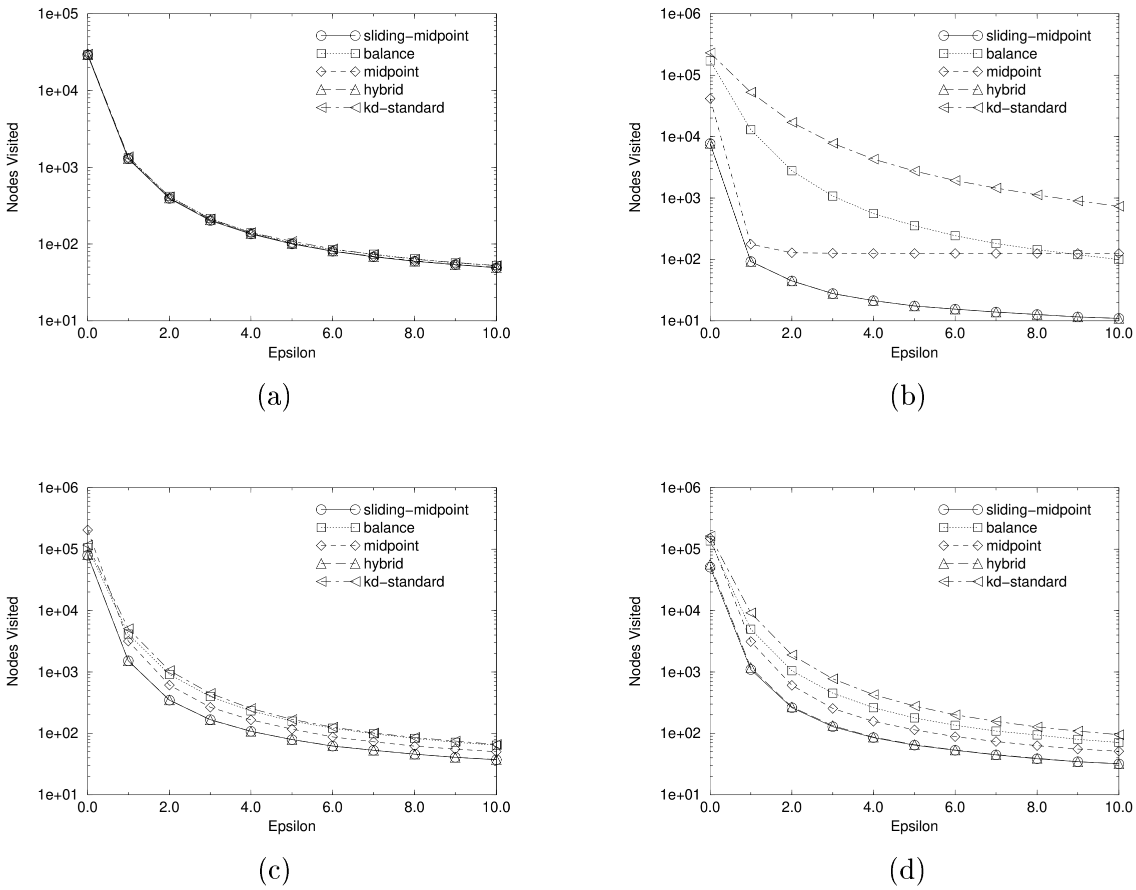

Figure 17: Average number of nodes visited for the (a) uniform, (b) clustered segments, (c) correlated Gaussian, and (d) correlated Laplacian distributions versus ϵ. Here $n = 128,000$.

degree?

Finally, it would be interesting to study other geometric query problems (e.g., point location and range searching) from the point of view of minimizing the expected query time. Recently, using techniques described in this paper (to be precise, Lemmas 10.3 and 10.4), we have made some progress on the planar point location problem [1].

13 Acknowledgements

We would like to thank Siu-Wing Cheng, Mordecai Golin, Ramesh Hariharan, David Mount, and Derick Wood for many useful comments and suggestions during the course of this work.

References

[1] S. Arya, S. W. Cheng, D. M. Mount, and H. Ramesh. Expected-case complexity of planar point location. Unpublished manuscript, 1999.

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

[3] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.

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

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

[6] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 1997.

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

[8] P. B. Callahan and S. R. Kosaraju. A decomposition of multi-dimensional point-sets with applications to k-nearest-neighbors and n-body potential fields. In *Proc. 24th Ann. ACM Sympos. Theory Comput.*, pages 546–556, 1992.

[9] T. Chan. Approximate nearest neighbor queries revisited. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 352–358, 1997.

[10] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 160–164, 1994.

[11] S. Deerwester, S. T. Dumals, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *J. Amer. Soc. Inform. Sci.*, 41:391–407, 1990.

[12] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, NY, 1973.

[13] 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.

[14] H. G. Eggleston. *Convexity.* Cambridge University Press, Cambridge, England, 1958.

[15] N. Farvardin and J. W. Modestino. Rate-distortion performance of DPCM schemes for autoregressive sources. *IEEE Transactions on Information Theory*, 31(3):402–418, May 1985.

[16] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: The QBIC system. *IEEE Computer*, 28:23–32, 1995.

[17] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.

[18] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic, Boston, MA, 1991.

[19] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, pages 604–613, 1998.

[20] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimension. In *Proc. 29th Annu. ACM Sympos. Theory Comput.*, pages 599–608, 1997.

[21] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimemsional spaces. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, 1998.

[22] S. Maneewongvatana and D. M. Mount. Analysis of approximate nearest neighbor searching with clustered point sets. In *ALENEX*, 1999.

[23] S. Maneewongvatana and D. M. Mount. It's okay to be skinny, if your friends are fat. 4th Annual CGC Workshop on Computational Geometry, URL: `ftp://ftp.cs.umd.edu/pub/faculty/mount/Papers/smpack.ps.gz`, 1999.

[24] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. 2nd Annual CGC Workshop on Computational Geometry, URL: `http://www.cs.umd.edu/~mount/ANN`, 1997.

[25] C. E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Journal*, 27:379–423, 623–656, 1948.