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. However, 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 perspective.

As a first step in this direction, we assume that the query points are sampled uniformly from a hypercube that encloses all the data points; however, we make no assumption on the distribution of the data points. We show that with a simple partition tree, called the sliding-midpoint tree, it is possible to achieve linear space and logarithmic 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, we prove that the sliding-midpoint tree achieves optimal expected query time in a certain class of algorithms.

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 for answering a query is more important than the worst-case time. Thus, when we have information on the query distribution, we believe it is prudent to incorporate it in the algorithm design with a view to minimize the *expected* query time and to provide simpler data structures. We study the approximate nearest neighbor problem from this novel perspective.

Nearest neighbor searching is a fundamental problem in computational geometry with applications in numerous areas such as pattern recognition [11], data compression [17], information retrieval [10], and multimedia databases [15]. Since the problem is very difficult to solve exactly in high dimensions, researchers have investigated the *approximate* nearest neighbor problem. Consider a set S of n data points in R^d and a query point $q \in R^d$. Given an error bound $\epsilon > 0$, we say that a point $p \in S$ is a $(1+\epsilon)$-*approximate nearest neighbor* of q

[*] This research was supported in part by RGC CERG HKUST736/96E and RGC DAG96/97.EG40. A preliminary version of this paper appeared in the *Proc. of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2000, 379–388.

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

if $dist(p,q) \leq (1+\epsilon)\,dist(p^*,q)$, where p^* is the true nearest neighbor of q. This problem has been extensively studied from the worst-case perspective [2, 3, 5, 6, 7, 8, 9, 12, 18, 19, 21].

For our expected-case study, we consider that the set S of n data points is contained within the unit hypercube $U = [0,1]^d$ and assume that the query points are sampled from the uniform distribution in U. Note that we make no assumption on the distribution of the data points. While the assumption of uniformly distributed query points is admittedly simplistic, we think it is a natural first step toward the design of algorithms for more general query distributions. We investigate a general approach for finding the approximate nearest neighbor devised by Arya et al. [4, 6], called *priority search*. This approach can be used in conjunction with any partition tree for S and is based on efficiently enumerating the leaves of the tree in order of increasing distance from the query point until a certain termination condition is satisfied. (Section 3 describes this method and presents its important features.) In this study, we consider the question of how to construct the partition tree so as to minimize the expected query time.

Our main results are for a partition tree based on a very simple splitting method called the *sliding-midpoint method*, introduced by Mount and Arya [24]. As in the standard kd-tree [16], cells are recursively subdivided using hyperplanes that are orthogonal to the coordinate-axes. More precisely, we first place a hyperplane orthogonal to the longest side of the cell and at its middle. We let this hyperplane be the splitting plane if there are data points located on both sides; otherwise, we slide the hyperplane toward the side that contains all the data points until it just touches a point. The point that touches the splitting plane is assigned to the side that is originally empty. (See Figure 1.) The space used by this tree is $O(n)$ (since no empty cells are created), and it can be easily constructed in $O(n \log n)$ time using well-known techniques [6]. Although its worst-case query time can be as bad as $\Omega(n)$, Maneewongvatana and Mount [22] have studied it empirically and observed that it performs well in practice. We present two results pertaining to its expected query time that provide some theoretical justification for its good practical performance.

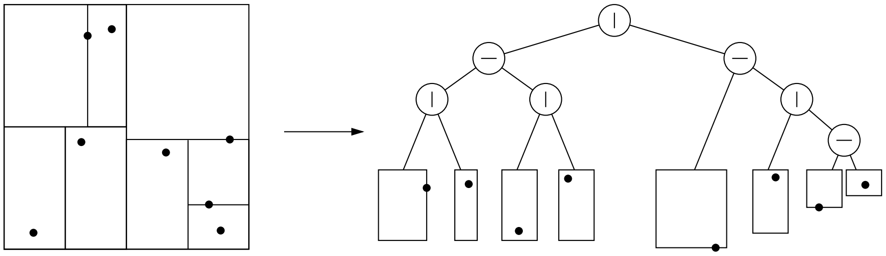


Figure 1: Sliding-midpoint tree.

First, we prove that the expected query time for this tree is $O((1/\epsilon)^d \log n)$, irrespective of the distribution of data points. We note that Arya et al. [6] and, more recently, Duncan, Goodrich, and Kobourov [12] have proposed partition trees for which priority search achieves logarithmic worst-case query time. However, they both use considerably more 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, Goodrich, and Kobourov restrict themselves to splits with a hyperplane; however, the hyperplane is not necessarily axis-orthogonal.) Although both these partition trees provide optimal worst-case query time, the features introduced to ensure this property hurt their practical performance.

Our second result, which is of greater significance, shows that the sliding-midpoint tree is, in fact, optimal in a certain sense. Consider the class of algorithms obtained by running priority search on partition trees, where the splits are made by hyperplanes at arbitrary orientations (that is, 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 modeled as an algebraic decision tree using linear tests (that is, involving the evaluation of a polynomial of degree one).

The paper is organized as follows. In Section 2, we describe our notation. Section 3 reviews the priority search approach for finding the approximate nearest neighbor. Section 4 presents a general framework for the expected-case analysis of partition trees. In Section 5, we give a simple proof that the expected query time using the sliding-midpoint tree is $O(\log n)$, and, in Section 6, we establish the optimality of this tree. Finally, we present our experimental results in Section 7.

2 Conventions

Let U denote the unit hypercube $[0,1]^d$ in d dimensions. We assume that the set S of n data points has been scaled and translated to lie within U. To avoid confusion, we always use *data point* to refer to a point in the data set S, and we use *point* to refer to any point in space. We assume that the dimension d and the error bound ϵ are fixed constants, independent of the number of data points.

We will assume that distances are measured in the Euclidean metric (although our results can be easily generalized to any Minkowski metric). We define the distance between a point p and a region z to be the minimum distance between p and any point in z.

Throughout the word *rectangle* will denote a d-dimensional axis-parallel hyperrectangle. We define the *size* of a rectangle to be the length of its longest side. For any region z, we let v_z denote its volume (area in two dimensions and length in one dimension). If z is the set-theoretic difference of two rectangles, one enclosed within the other, we denote the *outer rectangle* and *inner rectangle* by z_O and z_I, respectively, and we define its *size* to be the size of its outer rectangle.

The data structures we consider are based on partition trees that represent a hierarchical decomposition of U. We recall some basic facts about partition trees. Each node of the partition tree is associated with a region of space, called a *cell*, and the set of data points that lie in this cell. The cell associated with any node is partitioned into disjoint cells and associated with the children of the node. We assume that the cell associated with the root of the partition tree is the unit hypercube U. The leaves of the tree contain at most a constant number of data points, called the *bucket size* (henceforth assumed to be ≤ 1 for simplicity).

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. We let $\mathcal{Z}_T$ denote the subdivision of U induced by the leaf nodes of T. Let x be a node in T. We let C_x denote the cell associated with x and v_x denote the volume of C_x. If C_x is a rectangle or the difference of two rectangles, we use s_x to denote the size of C_x. We will often use R_x in place of C_x if C_x is a rectangle. Finally, we use ℓ_x to denote the level of x, that is, the length of the path from the root to x. Note that the root is at level 0.

3 Background

We briefly review some features of the priority search approach for approximate nearest neighbor searching, proposed by Arya and Mount [4] and Arya et al. [6].

3.1 Query Algorithm

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 point stored with the leaf and updates p, the closest neighbor to q found so far, and r_p, the distance to it. The algorithm terminates when the distance between q and the leaf cell exceeds $r_p/(1+\epsilon)$, returning p as the answer. For example, in Figure 2, 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 $r_p/(1+\epsilon)$ (the radius of the dotted circle shown).

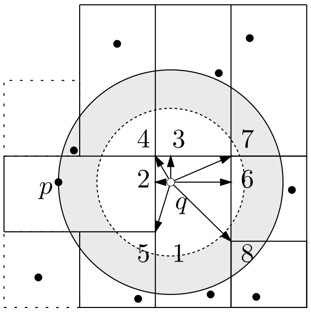


Figure 2: Algorithm overview.

To see that the algorithm works correctly for any partition tree, let B denote the ball of radius $r_p/(1+\epsilon)$ 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 of q.

It remains to describe the method used to enumerate the leaf cells in order of increasing distance from the query point. The algorithm maintains a priority queue of nodes, where the priority of a node is inversely related to the distance between the query point and the corresponding cell. Initially, the root of the tree is inserted into the queue. Then the following procedure is repeatedly carried out. First, the node with the highest priority is extracted from the queue, that is, the node 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 algorithm descends to the child that is closer to the query point, while the other child is inserted into the queue. When the algorithm reaches the leaf, it processes the data point associated with it. The correctness of the algorithm is based on the invariant that the set of leaves descended from the nodes in the priority queue is disjoint and their union is the set of all unvisited leaves.

While priority search can be applied on any partition tree, it is especially efficient on trees that use only axis-orthogonal splits. The reason is that, for each internal node visited, the algorithm needs to compute the distance between the query point and the children of the node. If we do not restrict ourselves to axis-orthogonal splits, this computation can take time proportional to the complexity of the cell. 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.* From a practical perspective, the difference is significant especially in high dimensions. Intuitively, this technique works because the cells for the children differ from the parent's cell in only one of the d dimensions. For details, we refer the reader to Arya and Mount [4].

3.2 Query Time

The analysis of the query time given in [6] employs the following two lemmas, which will also be useful for us. Lemma 3.1 says that all the leaf cells visited, except possibly the last one, are large relative to their distance from the query point. Lemma 3.2 relates the query time to the number of internal nodes and leaf nodes visited. Their proofs follow from the discussion in [6]; we repeat them here for the sake of completeness.

Lemma 3.1 *(see [6]) Consider a partition tree T built using axis-orthogonal splits, such that each leaf cell contains a data point. Suppose that we run priority search on T with query point q and error bound ϵ. Then the size of any leaf cell x that does not cause the algorithm to terminate is at least $r\epsilon/\sqrt{d}$, where r is the distance from q to x.*

Proof Suppose that the size of x is less than $r\epsilon/\sqrt{d}$. Clearly, the diameter of x is then less than $r\epsilon$, and so it contains a data point at distance less than $r(1+\epsilon)$ from q. This implies that x must satisfy the termination condition, which is a contradiction. □

Lemma 3.2 *(see [4, 6]) Consider a partition tree T built using axis-orthogonal splits. Suppose that we run priority search on T with query point q and error bound ϵ. Let I and L be the number of internal nodes and leaf nodes, respectively, visited by the algorithm. 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 On visiting an internal node, the algorithm first 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 priority queue; using Fibonacci heaps, the amortized time for each insertion is $O(1)$. Finally, it descends to the closer child, which takes $O(1)$ time.

On visiting a leaf node, it computes the distance between q and the data point stored with the leaf, which takes $O(d)$ time. It then extracts the closest node from the queue. Since each internal node visited inserts one child into the queue, the size of the queue is at most I. Thus it takes $O(\log I)$ time to extract the closest node. The total query time is therefore $O(I + L(d + \log I)) = O(I + Ld + L \log I)$. □

4 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 partition tree. For the sake of simplicity, we assume that the partition tree is built using axis-orthogonal splitting planes. Before presenting the analysis, we first need to make a small but important modification to priority search.

4.1 Modified Priority Search

We give some intuition for why this modification is needed. Recall from Section 3 that priority search is given a chance to terminate the search only at leaf nodes. Further, it follows from Lemma 3.1 that it visits at most one leaf cell of size less than $r\epsilon/\sqrt{d}$, where r is the distance from the query point to the leaf cell. However, while descending the tree to this small leaf cell, the algorithm may visit 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 modification is to allow the search to be terminated at internal nodes as well to ensure that at most one node (leaf or internal) of size less than $r\epsilon/\sqrt{d}$ is visited.

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 q. If $s_x \geq r\epsilon/\sqrt{d}$, the algorithm works just as before. However, if $s_x < r\epsilon/\sqrt{d}$, 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 a distance r from q, then the diameter of R_x is at most $r\epsilon$, and so the associated data point p_x is at a distance at most $r(1+\epsilon)$ from q. Note that the algorithm has already seen any data point that lies in the ball of radius r centered at q. Clearly, if this ball contains no data point, then p_x is a $(1+\epsilon)$-approximate nearest neighbor. Otherwise, the algorithm returns the nearest neighbor of q. 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. Note that Lemmas 3.1 and 3.2 continue to hold after this modification.

4.2 Upper Bound on Expected Number of Nodes Visited

Recall that S is a set of n data points in U and the query point is sampled from the uniform distribution in U. In Lemma 4.1, 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. Together with Lemma 3.2, these upper bounds facilitate the expected-case analysis of partition trees formed by different splitting methods. In Sections 5 and 6, we will apply them to the sliding-midpoint tree.

Lemma 4.1 *Let d and ϵ be any fixed constants. Let S be a set of n data points in U. 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 sampled from the uniform distribution in U. Then the expected number of internal nodes and leaf nodes visited by priority search is at most $1 + (1 + 2\sqrt{d}/\epsilon)^d \sum_{x \in \mathcal{I}_T} s_x^d$ and $1 + (1 + 2\sqrt{d}/\epsilon)^d \sum_{x \in \mathcal{L}_T} s_x^d$, respectively.*

Proof We will prove only the bound on the expected number of internal nodes visited, since the proof for leaf nodes is similar. Let q be a query point. Let x be any node visited that does not cause the algorithm to terminate, and let r be the distance between q and R_x. We claim that $s_x \geq r\epsilon/\sqrt{d}$. 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 3.1. Thus $r \leq s_x\sqrt{d}/\epsilon$. That is, if a node x is visited and does not cause the algorithm to terminate, then the query point q must be at a distance at most $s_x\sqrt{d}/\epsilon$ from R_x.

With each internal node x, associate a random variable V_x that 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 $I \leq 1 + \sum_{x \in \mathcal{I}_T} V_x$. 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,

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

Note that $E[V_x]$ equals the probability that V_x is 1. We compute an upper bound on this probability. From our earlier observation, it follows that if node x is visited and does not cause the algorithm to terminate, then q lies inside the hypercube of size $(1 + 2\sqrt{d}/\epsilon)s_x$, whose center coincides with the center of R_x. Since q is sampled from the uniform distribution in U, the volume of this hypercube, $(1+2\sqrt{d}/\epsilon)^d s_x^d$, is an upper bound on the desired probability and hence on $E[V_x]$. Using this bound in (1) completes the proof.

□

5 Logarithmic Bound on Expected Query Time of Sliding-Midpoint Tree

By Lemma 4.1, 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 two quantities for the sliding-midpoint tree.

Lemma 5.1 *Let S be a set of n data points in U. Let T be the partition tree for S built using the sliding-midpoint method. 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 depend on d.*

Proof Recall that the sliding-midpoint method 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 becomes 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.

For $i \geq 1$, define *block i* to consist of levels $(i-1)d$ to $id-1$. 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 $2^{(i-1)d}(2^d-1)$,

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

Let $\mathcal{I}'$ be the set of internal nodes in blocks numbered from 1 to $\lceil \log n/d \rceil$, and let $\mathcal{I}''$ 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 at most nd, which implies that

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

Therefore,

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

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. Inductively, assume that R'_x is the rectangle associated with node x. If x is an internal node, 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, $s_x \leq s'_x$, and for any leaf node x, $s_x \leq 2s'_x$, which implies that

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

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

$$\sum_{x \in \mathcal{L}_T} (s'_x)^d \leq 2^{d-1} \sum_{x \in \mathcal{L}_T} v'_x, \tag{3}$$

where v'_x denotes the volume of rectangle R'_x. Also, $\sum_{x \in \mathcal{L}_T} v'_x = 1$, since the rectangles R'_x associated with the leaves form a subdivision of U. Combining this with (2) and (3), we get $\sum_{x \in \mathcal{L}_T} s_x^d = O(1)$. □

Using the upper bounds obtained in Lemma 5.1, it is easy to bound the expected query time.

Theorem 5.1 *Let S be any set of n data points in $U = [0,1]^d$. Given any ϵ, assuming that the query point is sampled from the uniform distribution in U, the expected query time of priority search on the sliding-midpoint tree is $O((1/\epsilon)^d \log n)$. The constant factor in the O-notation depends on d.*

Proof By Lemma 3.2, 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 4.1 and 5.1, $E[I] = O((1/\epsilon)^d \log n)$ and $E[L] = O((1/\epsilon)^d)$. Thus the expected query time is $O((1/\epsilon)^d \log n)$. □

6 Optimality of Sliding-Midpoint Tree

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,it follows from Theorem 5.1 that 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 optimal performance for other data distributions as well. We prove that this is indeed the case under reasonable assumptions.

Let $\mathcal{T}_S$ denote the class of partition trees for S formed by splits using arbitrarily oriented hyperplanes (that is, *binary space partition trees*) such that each leaf cell contains at most one data point. We have the following result.

Theorem 6.1 *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 $U = [0,1]^d$, and let T be any partition tree in $\mathcal{T}_S$. Assuming that the query point is sampled from the uniform distribution in U, 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.*

Let $\mathcal{Z}$ denote any set of regions inside U. (Note that the regions need not form a subdivision of U, 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 U is defined to be the entropy of the set of regions that form the subdivision.

We give a brief overview of the proof. Let S be a set of n data points in U. Let $\mathcal{D}$ denote any set of cells in U 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 rectangles, one enclosed within the other. Note that the inner rectangle need not be present.

A.2. *Bounded aspect ratio:* The outer rectangle and inner rectangle (if present) have an aspect ratio (ratio of longest to shortest side) of at most 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:* There is a data point whose distance from any point inside the outer rectangle of the cell is at most $c_n s$, where s is the size of the cell (that is, the length of the longest side of its outer rectangle).

A.5. *Disjointedness:* 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 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).

6.1 Lower Bound

The main result of this subsection is the following lemma.

Lemma 6.1 *Let d and ϵ be any fixed constants. Let S be any set of n data points in U, and let T be any partition tree in $\mathcal{T}_S$. Let $\mathcal{D}$ be any set of cells in U satisfying properties* A.1–A.5. *Assuming that the query point is sampled from the uniform distribution in U, the expected query time of priority search on T is $\Omega(entropy(\mathcal{D}) + 1)$. The constant factor in the Ω-notation depends on d.*

Briefly, the proof works as follows. Let T be any partition tree in $\mathcal{T}_S$. Recall that $\mathcal{Z}_T$ denotes the subdivision of U induced by the leaf nodes of T. Since each internal node of T 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 $\mathcal{Z}_T$ either is 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 and satisfy the following properties for constant c_v, depending on dimension (property B.1 is obvious; property B.2 is proved in Lemma 6.2).

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 $z - (z \ominus B_r)$ (that is, the set of points inside z within distance r of the boundary of z) is at most $c_v r$.

Let $\mathcal{D}$ be any set of cells in U satisfying properties A.1–A.5, and let $\mathcal{Z}$ be any subdivision of U into cells satisfying properties B.1–B.2. Lemmas 6.3, 6.4, and 6.5 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 6.6. Finally, Lemma 6.7 proves that the expected query time of priority search on T is $\Omega(\mathit{entropy}(\mathcal{Z}_T) + 1)$. Together these imply 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 satisfies property B.2 (for constant $c_v = 2d$).

Lemma 6.2 *Let z be a convex polytope contained within U. 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 inward 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 sum of the volume of all the prisms, which is clearly $a_z r$, where a_z is the surface area (perimeter, in two dimensions) of z. Since $z \subseteq U$, a_z is no more than the surface area of U, 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 [13] for a proof.) Thus the volume of y is at most $2dr$. □

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.

Lemma 6.3 *Let d be any fixed constant. Let $\mathcal{Z}$ be any set of disjoint cells in U satisfying property B.2 (for constant c_v), and let $\mathcal{D}$ be any set of cells in U satisfying properties A.1, A.2 (for constant c_a), A.3, and A.5. Assume further that there exists a constant c_n such that, for any cell $u \in \mathcal{D}$ and $z \in \mathcal{Z}$, if $u_O \subseteq z$, then the distance to 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*

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

where the constant factor 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 will show that $\mathit{entropy}(\mathcal{F}_z) \leq dv_z \log(1/v_z) + O(v_z)$. Since $\mathit{entropy}(\mathcal{F}) = \sum_{z \in \mathcal{Z}} \mathit{entropy}(\mathcal{F}_z)$, the lemma will then follow by summing over all $z \in \mathcal{Z}$.

Let z be any cell in $\mathcal{Z}$. 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 a distance of 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 *subfragment* 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 subfragments. Since each fragment $y \in \mathcal{F}_z$ is partitioned into subfragments $y \cap z_i, i \geq 1$, it is easy to see that

$$\mathit{entropy}(\mathcal{F}_z) \leq \mathit{entropy}(\mathcal{G}_z). \tag{4}$$

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

$$\mathit{entropy}(\mathcal{G}_z) = \sum_{i \geq 1} \mathit{entropy}(\mathcal{G}_{z_i}). \tag{5}$$

Next we compute an upper bound on $\mathit{entropy}(\mathcal{G}_{z_i})$; the desired bound on $\mathit{entropy}(\mathcal{F}_z)$ will follow using (4) and (5). Obviously $\mathit{entropy}(\mathcal{G}_{z_i}) \leq \mathit{entropy}(\mathcal{G}_{z_i} \cup \{x\})$, where $x = z_i - \bigcup \mathcal{G}_{z_i}$ (that is, the region remaining in z_i after removing all the subfragments in $\mathcal{G}_{z_i}$). Let m_i be the number of subfragments in $\mathcal{G}_{z_i}$. It follows from basic calculus that $\mathit{entropy}(\mathcal{G}_{z_i} \cup \{x\})$ can be no more than the entropy of the set of regions formed by splitting z_i into $m_i + 1$ parts of equal volume. Thus

$$\mathit{entropy}(\mathcal{G}_{z_i}) \leq v_{z_i} \log \frac{m_i + 1}{v_{z_i}}. \tag{6}$$

We will show that $m_i \leq c \cdot 2^{id}/v_z^{d-1}$, 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

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

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

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

Using (5), we obtain

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

where the constant factor in the O-notation depends on d. By (4), $\mathit{entropy}(\mathcal{F}_z) \leq dv_z \log(1/v_z) + O(v_z)$, which is the desired claim.

It remains to show that $m_i \leq c \cdot 2^{id}/v_z^{d-1}$. 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 $c_n' = \max(c_n, \sqrt{d})$. Below we describe a set of disjoint hypercubes $\mathcal{H}$ and give a 1-1 mapping from $\mathcal{H}$ to $\mathcal{F}_{z_i'}$ such that at least a fraction $(1/2d)$ of the fragments in $\mathcal{F}_{z_i'}$ lie in the image of this mapping. (We will use H_y to denote the hypercube, if any, that maps to fragment y and say that H_y is *associated* with y.) Further, each hypercube in $\mathcal{H}$ is contained within z and has size $r_i/(c_n' c_a)$. A simple packing argument then implies that

$$|\mathcal{H}| \leq \frac{v_z}{(r_i/(c_n' c_a))^d}.$$

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

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

The number of fragments in $\mathcal{F}_{z_i'}$ is at most $2d$ times this quantity. We thus obtain the desired bound $m_i \leq c \cdot 2^{id}/v_z^{d-1}$, 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 $r_i/\sqrt{d}$. Otherwise, u_O is contained within z, and, by the statement of the lemma, the distance to 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 a distance of at least r_i from the boundary of z; thus $c_n s_u \geq r_i$, which implies that $s_u \geq r_i/c_n$. Combining the two cases, we get $s_u \geq r_i/c_n'$, where $c_n' = \max(c_n, \sqrt{d})$. We will make use of this fact in our proof.

We now describe how to associate hypercubes with fragments in $\mathcal{F}_{z_i'}$; later we will show that the resulting set of hypercubes $\mathcal{H}$ satisfies all the properties mentioned earlier. 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 representative 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 of size at least r_i/c_n', and (iii) u has an inner rectangle of size less than r_i/c_n'.

If u has no inner rectangle, then, with each fragment $y \in \mathcal{F}_{z_i'}$ such that $y \subseteq u$, associate 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, as shown above, $s_u \geq r_i/c_n'$ 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 at least 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, Figures 3(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 representative points corresponding to the fragments in $\mathcal{F}_{z'_i}$. With each fragment $y \in \mathcal{F}_{z'_i}$ such that $p_y \in R$, associate a hypercube H_y of size $r_i/(c'_n c_a)$ that overlaps p_y and is contained within R. With the remaining fragments in u (that is, fragments whose corresponding representative point lies in $u - R$), we associate no hypercube.

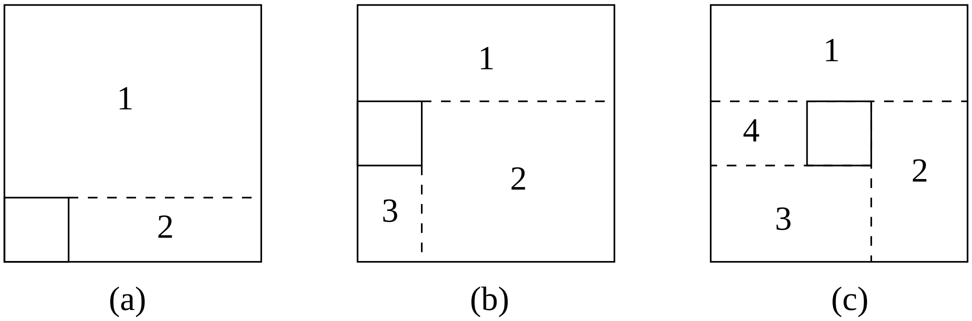


Figure 3: Partitioning a cell into rectangles.

In the third case, u has an inner rectangle of size less than 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 representative points corresponding to the fragments in $\mathcal{F}_{z'_i}$. With each fragment $y \in \mathcal{F}_{z'_i}$ such that $p_y \in R$, associate 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. Such a hypercube exists because, as shown above, $s_u \geq r_i/c'_n$. With the remaining fragments in u, we associate no hypercube. (Note that, unlike the second case, the sides of R may have length less than $r_i/(c'_n c_a)$, 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 $u_1, u_2 \in \mathcal{D}$ be the cells containing y_1, 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) $u_1 = u_2$. (Note that, in Case (iv), 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 representative points lying in only one of these rectangles.) Since y_1 and y_2 are distinct fragments created by the intersection of z with u_1, 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. □

Lemma 6.4 *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* 6.3*. Further, let $\mathcal{F}$ be as defined in the statement of Lemma* 6.3*. Assuming that $\mathcal{Z}$ is a subdivision of U,*

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

where the constant factor 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 6.3 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. □

Lemma 6.5 *Let d be any fixed constant. Let S be any set of n data points in U. Let $\mathcal{Z}$ be a subdivision of U whose cells satisfy properties B.1–B.2, and let $\mathcal{D}$ be a set of cells in U satisfying properties A.1–A.5. Then*

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

where the constant factor 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 the distance to any point p in u_O from the boundary of z is at most $c_n s_u$. This follows from the facts that the distance between p 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 6.4, which implies the desired claim. □

Lemma 6.6 *Let d be any fixed constant. Let S be any set of n data points in U, and let T be any partition tree in $\mathcal{T}_S$. Then $\mathit{entropy}(\mathcal{D}) = O(\mathit{entropy}(\mathcal{Z}_T) + 1)$, where $\mathcal{D}$ is any set of cells in U satisfying properties A.1–A.5. The constant factor in the O-notation depends on d.*

Proof Refine the subdivision $\mathcal{Z}_T$ by splitting each nonempty 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 $\mathit{entropy}(\mathcal{Z}'_T)$ 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

$$\mathit{entropy}(\mathcal{Z}'_T) \leq \sum_{z \in \mathcal{Z}_T} 2\frac{v_z}{2} \log \frac{1}{v_z/2} = \sum_{z \in \mathcal{Z}_T} \left(v_z \log \frac{1}{v_z} + v_z \right) = \mathit{entropy}(\mathcal{Z}_T) + 1, \qquad (7)$$

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 into cells satisfying property B.1 (empty interior). Also, since the cells in $\mathcal{Z}'_T$ are convex polytopes contained within U, by Lemma 6.2, they satisfy property B.2 (proportionality of swept volume to radius). Thus $\mathcal{D}$ and $\mathcal{Z}'_T$ satisfy the conditions of Lemma 6.5, which implies that

$$\mathit{entropy}(\mathcal{D}) = O(\mathit{entropy}(\mathcal{Z}'_T) + 1). \tag{8}$$

The lemma now follows from (7) and (8). □

Lemma 6.7 *Let d and ϵ be any fixed constants. Let S be any set of n data points in U, and let T be any partition tree in $\mathcal{T}_S$. Assuming that the query point is sampled from the uniform distribution in U, the expected query time of priority search on T is $\Omega(\mathit{entropy}(\mathcal{Z}_T) + 1)$.*

Proof Recall that the algorithm starts by descending from the root of the tree T 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 $\sum_{z \in \mathcal{Z}_T} v_z(\ell_z + 1)$. Note that this is the weighted external path length [20] of the tree, where the weight of a leaf is the volume of the associated cell. A fundamental information theoretic result due to Shannon [20, 25] implies that the weighted external 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 $\mathit{entropy}(\mathcal{Z}_T)$ 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 6.6 and 6.7 together imply Lemma 6.1.

6.2 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 $O(\mathit{entropy}(\mathcal{D}) + 1)$, where $\mathcal{D}$ is some set of cells in U satisfying properties A.1–A.5. In view of Lemma 6.1, this would imply the optimality of the sliding-midpoint tree in the sense of Theorem 6.1.

Our approach for obtaining $\mathcal{D}$ is as follows. In Lemma 6.8, we show that, for any sliding-midpoint tree T, there exists a closely related partition tree T' whose leaves satisfy properties A.1–A.5 and whose internal nodes satisfy properties A.1–A.4. In addition, T' possesses 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 level is $O(\log(1/v) + 1)$.

We define $\mathcal{D}$ to be the set of cells corresponding to the leaves of T'. In Lemmas 6.9, 6.10, 6.11, and 6.12, by exploiting properties of T and T', we give a simple analysis proving that the expected query time of priority search on T is $O(\mathit{entropy}(\mathcal{D}) + 1)$.

Lemma 6.8 *Let S be a set of n data points in U. Let T be the sliding-midpoint tree for S. Then there exists a partition tree T' such that the following hold:*

(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 u by R_u if it is a rectangle and by C_u 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. 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 labeling implies that x is a leaf in T_{i-1}. Thus, by (4(c)) of the 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 that 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.

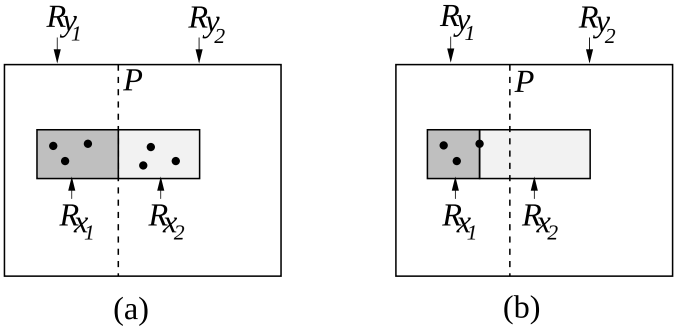


Figure 4: Proof of Lemma 6.8.

Case 1: P is the splitting plane associated with node x. (See Figure 4(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 substeps and show that the invariant holds after the second substep.

Substep 1: By (4(a)) of the invariant, the cell associated with y is a rectangle R_y, and $R_x \subseteq R_y$. We create two children y_1 and y_2 for node y. We then 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, it trivially satisfies properties A.1 (difference of two rectangles) and A.3 (stickiness). Second, since $R_{x_1} \subseteq R_{y_1}$ and there is a data point in R_{x_1}, it is clear that 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 (4(b)) of the 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 (that is, the 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 the 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 7/8th the volume of R_y. As we will see, substep 2 creates either zero or two children for y_1; thus the volume of the nodes decreases by at least a factor of 8/7 after every two levels of descent in T'_i, which implies that T'_i satisfies property A.6 at the end of substep 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.

Substep 2: Let $B_1 \subseteq R_{y_1}$ denote a rectangle formed by expanding each side of R_{x_1}

that is smaller than $s_{x_1}/2$ to $s_{x_1}/2$. 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, as shown in the discussion of substep 1, each side of R_{y_1} is of length $\geq s_x/2 \geq s_{x_1}/2$. Clearly, 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 (since its length can increase by a factor of at most 2 in each of two expansions). Thus the length of each side of B_2 is $\geq s_{x_1}/2$ and $\leq 4s_{x_1}$.

If $B_2 = R_{y_1}$, then set $f(x_1) = y_1$. Otherwise, create two children 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. 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 substep 1. We next 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 children are added, and there is nothing to show. 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 properties A.1 (difference of two rectangles) and A.2 (bounded aspect ratio). Since B_2 is a rectangle, it satisfies property A.3 (stickiness). By construction of B_2, it is clear that $C_{y_{12}}$ also satisfies stickiness. Since B_2 has a data point inside it, it follows that B_2 and $C_{y_{12}}$ both satisfy property A.4 (existence of a close data point).

Next we show that (4(a)–(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 (4(a)), note that by construction $R_{f(x_1)}$ is the rectangle B_2, 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 (4(b)) and (5). Since $f(x_1)$ is a leaf in the tree T'_i, it follows that (4(c)) holds.

Finally, it is clear from our construction that T'_i is a partition tree whose leaves satisfy property A.5, and f is a 1-1 mapping from the nodes of T_i to the nodes of T'_i ((1) and (3) of the invariant).

Case 2: P is not the splitting plane associated with node x. (See Figure 4(b).)

This implies that all the data points in R_x are on the same side of P, and the splitting plane 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 4(b). (The other case can be handled similarly.) Note that x_2 is a leaf containing exactly one data point.

As in Case 1, the cell associated with node y is a rectangle R_y and $R_x \subseteq R_y$. We modify T'_{i-1} in two substeps. In substep 1, we create two children y_1 and y_2 for node y. We then split R_y into two rectangles by hyperplane P; the rectangle that 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 substep 2 is identical to Case 1, and so we omit it. Note that we need only to process nodes x_1 and y_1 in substep 2 (since x_2 is already mapped by f in substep 1).

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

Arguing as in Case 1, we can show that the length of each side of R_{y_2} is $\geq s_x/2$ and $\leq 4s_x$. Further, since R_x contains a data point (call it p), and $R_x \subseteq R_y$, the distance between any point in R_y and p is no more than the diameter of R_y; by (4(b)) of the invariant, this is $\leq 4\sqrt{d}s_x$. Since $R_{y_2} \subseteq R_y$, this bound also applies to the distance between any point in R_{y_2} and p. It follows that R_{y_2} satisfies property A.4. Finally, to show the desired lower and upper bound on the size of R_{y_2}, note that R_{x_2} is the larger of the two children formed by applying the sliding-midpoint method to R_x. Thus $s_x \geq s_{x_2} \geq s_x/2$, which implies that the length of each side of R_{y_2} is $\geq s_{x_2}/2$ and $\leq 8s_{x_2}$. This completes the proof.

□

The following lemma proved in [23] gives a bound on the number of leaf cells visited by priority search that holds irrespective of the data distribution and the location of the query point.

Lemma 6.9 *(see [23]) The number of leaf cells of the sliding-midpoint tree visited by priority search in the worst case is $O((1+1/\epsilon)^d)$. The constant factor in the O-notation depends on d.*

Let T denote the sliding-midpoint tree, and let T' denote the tree corresponding to it, given in the statement of Lemma 6.8.

Lemma 6.10 *The expected query time of priority search on the sliding-midpoint tree T is $O(\sum_{x \in \mathcal{N}_{T'}} s_x^d)$. The constant factor in the O-notation depends on d and ϵ.*

Proof By Lemma 3.2, 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, the expected query time is $O(E[I] + E[L \log I])$. By Lemma 6.9, L is bounded by a constant. Hence the expected query time is $O(E[I] + E[\log I]) = O(E[I])$. Lemma 4.1 implies that $E[I] = O(\sum_{x \in \mathcal{N}_T} s_x^d)$. By Lemma 6.8, there is a 1-1 function f that maps each node in T to a node in T', whose size is the same to within a constant factor. The lemma now follows. □

Lemma 6.11 *Let $\tilde{T}$ be any partition tree in which the cells associated with the leaf and internal nodes of the tree satisfy properties A.1–A.3 and A.6. Let $\mathcal{Z}_{\tilde{T}}$ denote the subdivision of U induced by the leaf nodes of $\tilde{T}$. Then $\sum_{x \in \mathcal{N}_{\tilde{T}}} s_x^d = O(\mathit{entropy}(\mathcal{Z}_{\tilde{T}}) + 1)$, where the constant factor in the O-notation depends on d.*

Proof Since the cells associated with the nodes of the tree satisfy property A.2 (bounded aspect ratio) and property A.3 (stickiness), it follows that $\sum_{x \in \mathcal{N}_{\tilde{T}}} s_x^d = O\left(\sum_{x \in \mathcal{N}_{\tilde{T}}} v_x\right)$. Since the volume of a node x is the sum of the volume of all the leaf nodes descended from it, we can write $\sum_{x \in \mathcal{N}_{\tilde{T}}} v_x = \sum_{x \in \mathcal{L}_{\tilde{T}}} v_x(\ell_x + 1)$, where ℓ_x denotes the level of leaf x. Further, by property A.6, $\ell_x = O(\log(1/v_x) + 1)$. Thus $\sum_{x \in \mathcal{N}_{\tilde{T}}} s_x^d = O\left(\sum_{x \in \mathcal{L}_{\tilde{T}}} v_x\left(\log(1/v_x) + 1\right)\right) =$

$O(\mathit{entropy}(\mathcal{Z}_{\tilde{T}}) + 1)$. □

By Lemma 6.8, T' satisfies the conditions of Lemma 6.11, and thus $\sum_{x \in \mathcal{N}_{T'}} s_x^d = O(\mathit{entropy}(\mathcal{Z}_{T'}) + 1)$. Setting $\mathcal{D} = \mathcal{Z}_{T'}$ and applying Lemmas 6.8 and 6.10, we obtain the desired upper bound on the expected query time.

Lemma 6.12 *The expected query time of priority search on the sliding-midpoint tree is $O(\mathit{entropy}(\mathcal{D}) + 1)$, where $\mathcal{D}$ is some set of cells in U satisfying properties A.1–A.5. The constant factor in the O-notation depends on d and ϵ.*

Finally, combining this lemma with the lower bound given in Lemma 6.1 establishes Theorem 6.1.

Remark: It is easy to see that our proof of Theorem 6.1 also implies the following: Let S be any set of n data points in U, and let A be any algorithm for finding the approximate nearest neighbor that can be modeled as an algebraic decision tree T using linear 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 mention only two key observations. First, the leaf nodes of T induce a subdivision $\mathcal{Z}_T$ of U 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 6.1, we can prove that the expected query time of A is $\Omega(\mathit{entropy}(\mathcal{D}) + 1)$, where $\mathcal{D}$ is any set of cells in U satisfying properties A.1–A.5. The claim now follows in light of Lemma 6.12.

7 Experimental Results

We ran experiments to compare the performance of the sliding-midpoint tree with the *standard kd-tree*, a partition tree devised by Friedman, Bentley, and Finkel [16], which is often used in nearest neighbor searching (both exact and approximate). The standard kd-tree recursively splits the data set into two sets of equal size by a hyperplane orthogonal to the dimension in which the points have maximum *spread* (difference of maximum and minimum coordinate). Friedman, Bentley, and Finkel showed that this tree can be used to answer nearest neighbor queries in $O(\log n)$ expected time, assuming that both data and query points are sampled from a distribution of bounded density.

We start by listing the point distributions used for generating the data sets [6, 24]. The clustered segments distribution was used to model data sets that 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 [14] for more information.

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.

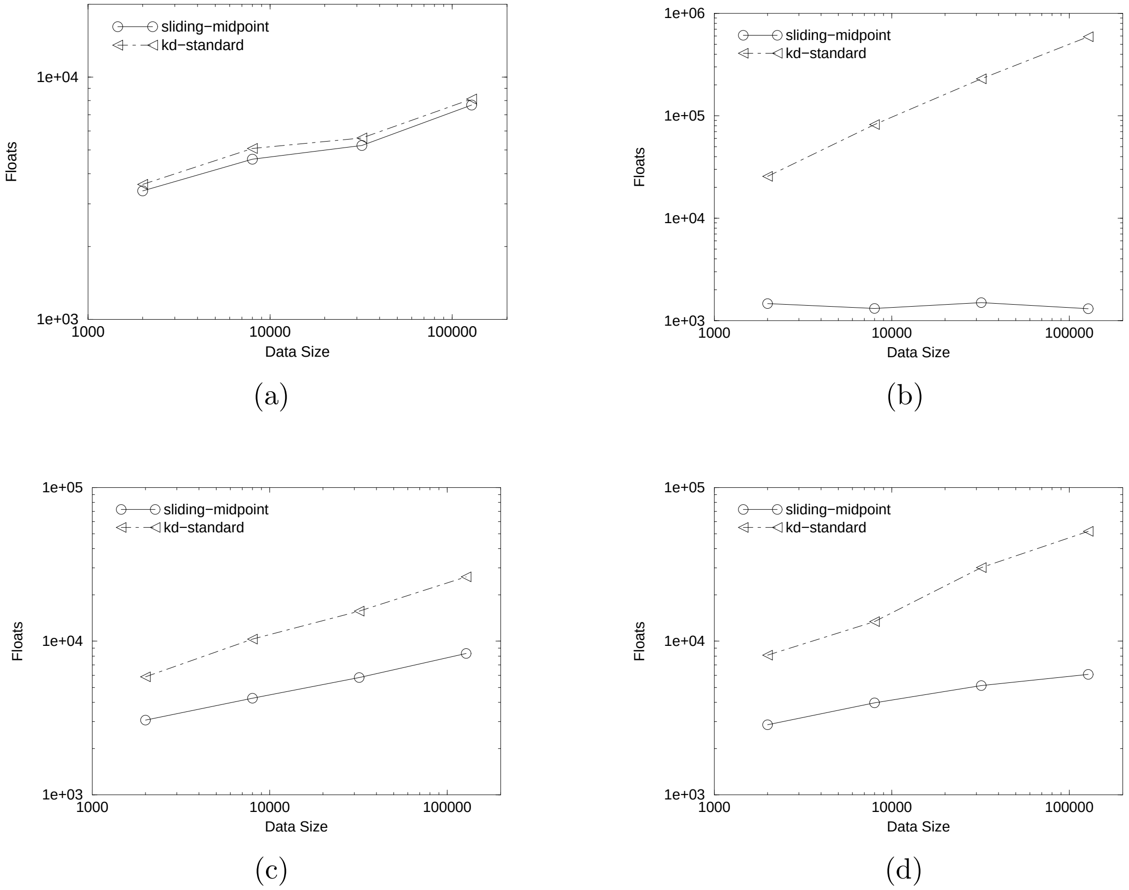
We generated data points in dimension 16 from these distributions and in each case generated query points uniformly from a hypercube enclosing 90% of the data points (to reduce the effect of outliers). Throughout we used a bucket size (the maximum number of data points inside a leaf cell) of one for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

For each experiment, we fixed ϵ and measured a number of statistics, averaging over 200 query points. The statistics included the average number of nodes visited, the average number of floating point operations (that is, any arithmetic operation involving point coordinates or distances), and the average CPU time. We present plots showing the number of floating point operations, which agrees well with the CPU times and provides a reasonable machine-independent measure of the query time.

The results for the uniform, clustered segments, correlated Gaussian, and correlated Laplacian distributions are shown in Figures 5 and 6. Figure 5 shows the average number of floating point operations as a function of n when ϵ is 2. Figure 6 shows the average number of floating point operations as a function of ϵ when n is $128,000$. We used a large value of ϵ in some of our experiments because we observed that the actual relative error committed by the algorithm is typically smaller by factors between 10 and 100.

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 offers significant speed-up, sometimes by factors of over 10, compared to the standard kd-tree. Moreover, the speed-up increases with n.



(a)

(b)

(c)

(d)

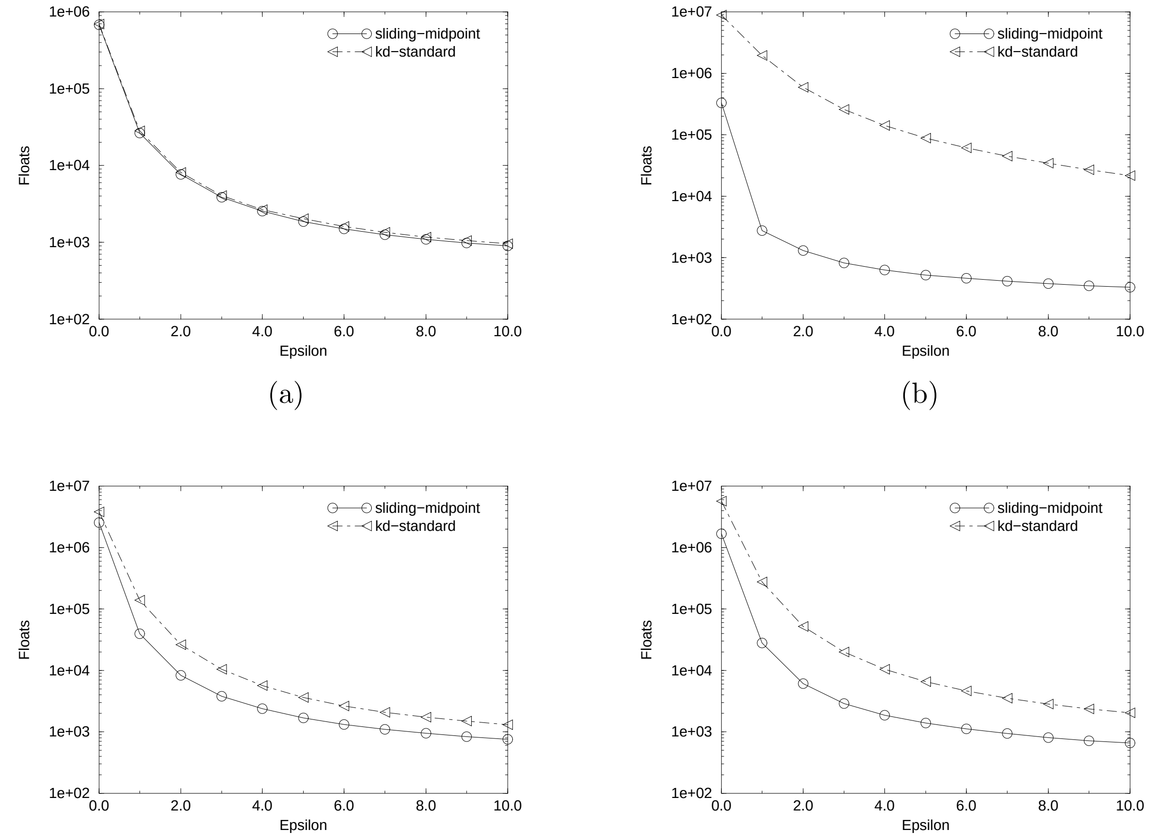
Figure 5: Average number of floating point operations for the (a) uniform, (b) clustered segments, (c) correlated Gaussian, and (d) correlated Laplacian distributions versus n. Here $\epsilon = 2$.

- For the uniform distribution, both trees yield very similar query times.
- For the clustered segments distribution, the expected query time for the sliding-midpoint tree appears to be independent of the number of data points. (This is not hard to explain using Lemma 4.1.)
- As ϵ increases from 0 to 2, the query times of both the trees decrease significantly, usually by factors between 10 and 100.

Because of its design, the sliding-midpoint tree does a better job than the standard kd-tree of zooming toward the region where the data points are more densely clustered. This is the reason why it enjoys a considerable advantage for clustered data sets.

8 Conclusion

We have studied the approximate nearest neighbor problem from the perspective of expected-case performance. Our analysis assumes that the query points are sampled uniformly from a



(a) (b)

(c) (d)

Figure 6: Average number of floating point operations for the (a) uniform, (b) clustered segments, (c) correlated Gaussian, and (d) correlated Laplacian distributions versus ϵ. Here $n = 128,000$.

hypercube enclosing all the data points but makes no assumption on the distribution of data points. We have shown that the sliding-midpoint tree achieves linear space and logarithmic expected query time. We have also shown that this tree attains optimal expected query time (ignoring constant factors) for any set of data points in a certain class of algorithms. The data structure is simple and easy to implement, and our empirical studies indicate that it performs well in practice.

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 nonuniform query distributions. Another problem concerns strengthening the optimality claim for the sliding-midpoint tree. We proved that the sliding-midpoint tree achieves expected query time no more than a constant times that of any algorithm that can be modeled as an algebraic decision tree using linear tests. Does this optimality claim hold even if we allow polynomials of higher degree?

9 Acknowledgements

We would like to thank Siu-Wing Cheng, Mordecai Golin, Ramesh Hariharan, David Mount, Derick Wood, and the anonymous referees for many useful comments and suggestions.

References

[1] S. Arya, S.-W. Cheng, D. M. Mount, and H. Ramesh. Efficient expected-case algorithms for planar point location. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Comput. Sci. 1851, pages 353–366. Springer-Verlag, 2000.

[2] S. Arya and T. Malamatos. Linear-size approximate Voronoi diagrams. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 147–155, 2002.

[3] S. Arya, T. Malamatos, and D. M. Mount. Space-efficient approximate Voronoi diagrams. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 721–730, 2002.

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

[5] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, 1993.

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

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

[8] T. M. Chan. Approximate nearest neighbor queries revisited. *Discrete Comput. Geom.*, 20:359–373, 1998.

[9] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proceedings of the 10th Annual ACM Symposium on Computional Geometry*, pages 160–164, 1994.

[10] 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.

[11] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* John Wiley, New York, 1973.

[12] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. *J. Algorithms*, 33:303–333, 2001.

[13] H. G. Eggleston. *Convexity.* Cambridge University Press, Cambridge, UK, 1958.

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

[15] 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.

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

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

[18] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[19] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimension. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 599–608, 1997.

[20] D. E. Knuth. *Sorting and Searching.* The Art of Computer Programming 3. Addison-Wesley, Reading, MA, second edition, 1998.

[21] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimemsional spaces. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 614–623, 1998.

[22] S. Maneewongvatana and D. M. Mount. Analysis of approximate nearest neighbor searching with clustered point sets. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation*, 1999.

[23] S. Maneewongvatana and D. M. Mount. It's okay to be skinny, if your friends are fat. In *Proceedings of the 4th Annual CGC Workshop on Computational Geometry*, 1999.

[24] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. In *Proceedings of the 2nd Annual CGC Workshop on Computational Geometry*, 1997.

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