An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions*

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

July 6, 1998

Abstract

Consider a set S of n data points in real d-dimensional space, R^d, where distances are measured using any Minkowski metric. In nearest neighbor searching we preprocess S into a data structure, so that given any query point $q \in R^d$, the closest point of S to q can be reported quickly. Given any positive real ϵ, a data point p is a $(1+\epsilon)$-*approximate nearest neighbor* of q if its distance from q is within a factor of $(1+\epsilon)$ of the distance to the true nearest neighbor. We show that it is possible to preprocess a set of n points in R^d in $O(dn \log n)$ time and $O(dn)$ space, so that given a query point $q \in R^d$, and $\epsilon > 0$, a $(1+\epsilon)$-approximate nearest neighbor of q can be computed in $O(c_{d,\epsilon} \log n)$ time, where $c_{d,\epsilon} \leq d \lceil 1 + 6d/\epsilon \rceil^d$ is a factor depending only on dimension and ϵ. In general, we show that given an integer $k \geq 1$, $(1+\epsilon)$-approximations to the k nearest neighbors of q can be computed in additional $O(kd \log n)$ time.

Key words: Nearest neighbor searching, post-office problem, closest-point queries, approximation algorithms, box-decomposition trees, priority search.

1 Introduction.

Nearest neighbor searching is the following problem: we are given a set S of n *data points* in a metric space, X, and the task is to preprocess these points so that, given any *query point* $q \in X$, the data point nearest to q can be reported quickly. This is also called the *closest-point problem*

*A preliminary version of this paper appeared in the *Proc. of the Fifth Annual ACM-SIAM Symp. on Discrete Algorithms*, 1994, pp. 573–582.

†Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. The work of this author was partially supported by HK RGC grant HKUST 736/96E. Part of this research was conducted while the author was visiting the Max-Planck-Institut für Informatik, Saarbrücken, Germany. Email: `arya@cs.ust.hk`

‡Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland. The support of the National Science Foundation under grant CCR–9712379 is gratefully acknowledged. Email: `mount@cs.umd.edu`

§Center for Automation Research, University of Maryland, College Park, Maryland, and Space Data and Computing Division, NASA Goddard Space Flight Center, Greenbelt, Maryland. This research was carried out, in part, while the author held a National Research Council NASA Goddard Associateship. Email: `nathan@cfar.umd.edu`

¶Department of Computer Science, University of the District of Columbia, Washington, DC, and Center for Automation Research, University of Maryland, College Park, Maryland. The support of the National Science Foundation under grant CCR–9310705 is gratefully acknowledged. Email: `ruth@cfar.umd.edu`

‖Department of Computer Science and Information Systems, The American University, Washington, DC. Email: `awu@american.edu`

and the *post office problem*. Nearest neighbor searching is an important problem in a variety of applications, including knowledge discovery and data mining [FPSSU96], pattern recognition and classification [CH67, DH73], machine learning [CS93], data compression [GG91], multimedia databases [FSN+95], document retrieval [DDF+90], and statistics [DW82].

High-dimensional nearest neighbor problems arise naturally when complex objects are represented by vectors of d numeric features. Throughout we will assume the metric space X is real d-dimensional space R^d. We also assume distances are measured using any Minkowski L_m distance metric. For any integer $m \geq 1$, the *L_m-distance* between points $p = (p_1, p_2, \ldots, p_d)$ and $q = (q_1, q_2, \ldots, q_d)$ in R^d is defined to be the m-th root of $\sum_{1 \leq i \leq d} |p_i - q_i|^m$. In the limiting case, where $m = \infty$, this is equivalent to $\max_{1 \leq i \leq d} |p_i - q_i|$. The L_1, L_2, and L_∞ metrics are the well-known Manhattan, Euclidean and max metrics, respectively. We assume that the distance between any two points in R^d can be computed in $O(d)$ time. (Note that the root need not be computed when comparing distances.) Although this framework is strong enough to include many nearest neighbor applications, it should be noted that there are applications that do not fit within this framework (e.g., computing the nearest neighbor among strings, where the distance function is the edit distance, the number of single character changes).

Obviously the problem can be solved in $O(dn)$ time through simple brute-force search. A number of methods have been proposed which provide relatively modest constant factor improvements (e.g., through partial distance computation [BG85], or by projecting points onto a single line [FBS75, GK92, LC94]). Our focus here is on methods using data structures that are stored in main memory. There is a considerable literature on nearest neighbor searching in databases. For example, see [BBKK97, BKK96, LJF94, RKV95, WJ96].

For uniformly distributed point sets, good expected case performance can be achieved by algorithms based on simple decompositions of space into regular grids. Rivest [Riv74] and later Cleary [Cle79] provided analyses of these methods. Bentley, Weide, and Yao [BWY80] also analyzed a grid-based method for distributions satisfying certain bounded-density assumptions. These results were generalized by Friedman, Bentley, and Finkel [FBF77] who showed that $O(n)$ space and $O(\log n)$ query time are achievable in the expected case through the use of kd-trees. However, even these methods suffer as dimension increases. The constant factors hidden in the asymptotic running time grow at least as fast as 2^d (depending on the metric). Sproull [Spr91] observed that the empirically measured running time of kd-trees does increase quite rapidly with dimension. Arya, et al. [AMN95] showed that if n is not significantly larger than 2^d, as arises in some applications, then boundary effects mildly decrease this exponential dimensional dependence.

From the perspective of worst-case performance, an ideal solution would be to preprocess the points in $O(n \log n)$ time, into a data structure requiring $O(n)$ space so that queries can be answered in $O(\log n)$ time. In dimension 1 this is possible by sorting the points, and then using binary search to answer queries. In dimension 2, this is also possible by computing the Voronoi diagram for the point set and then using any fast planar point location algorithm to locate the cell containing the query point. (For example, see [dBvKOS97, Ede87, PS85].) However, in dimensions larger than 2, the worst-case complexity of the Voronoi diagram grows as $O(n^{\lceil d/2 \rceil})$. Higher dimensional solutions with sublinear worst-case performance were considered by Yao and Yao [YY85]. Later Clarkson [Cla88] showed that queries could be answered in $O(\log n)$ time with $O(n^{\lceil d/2 \rceil + \delta})$ space, for any $\delta > 0$. The O-notation hides constant factors that are exponential in d. Agarwal and Matoušek [AM93a] generalized this by providing a tradeoff between space and query time. Meiser [Mei93] showed that exponential factors in query time could be eliminated by giving an algorithm with $O(d^5 \log n)$ query time and $O(n^{d+\delta})$ space, for any $\delta > 0$. In any fixed dimension greater than 2, no known method achieves the simultaneous goals of roughly linear space and logarithmic query

time.

The apparent difficulty of obtaining algorithms that are efficient in the worst case with respect to both space and query time for dimensions higher than 2, suggests that the alternative approach of finding *approximate* nearest neighbors is worth considering. Consider a set S of data points in R^d and a query point $q \in R^d$. Given $\epsilon > 0$, we say that a point $p \in S$ is a $(1+\epsilon)$-*approximate nearest neighbor* of q if

$$dist(p,q) \leq (1+\epsilon)dist(p^*,q),$$

where p^* is the true nearest neighbor to q. In other words, p is within relative error ϵ of the true nearest neighbor. More generally, for $1 \leq k \leq n$, a kth $(1+\epsilon)$-approximate nearest neighbor of q is a data point whose relative error from the true kth nearest neighbor of q is ϵ. For $1 \leq k \leq n$, define a *sequence* of k approximate nearest neighbors of query point q to be a sequence of k distinct data points, such that the ith point in the sequence is an approximation to the ith nearest neighbor of q. Throughout we assume that both d and ϵ are fixed constants, independent of n, but we will include them in some of the asymptotic results to indicate the dependency on these values.

The approximate nearest neighbor problem has been considered by Bern [Ber93]. He proposed a data structure based on quadtrees, which uses linear space and provides logarithmic query time. However, the approximation error factor for his algorithm is a fixed function of the dimension. Arya and Mount [AM93c] proposed a randomized data structure which achieves polylogarithmic query time in the expected case, and nearly linear space. In their algorithm the approximation error factor ϵ is an arbitrary positive constant, fixed at preprocessing time. In this paper, we strengthen these results significantly. Our main result is stated in the following theorem.

Theorem 1 *Consider a set S of n data points in R^d. There is a constant $c_{d,\epsilon} \leq d\lceil 1+6d/\epsilon \rceil^d$, such that in $O(dn \log n)$ time it is possible to construct a data structure of size $O(dn)$, such that for any Minkowski metric:*

(i) Given any $\epsilon > 0$ and $q \in R^d$, a $(1+\epsilon)$-approximate nearest neighbor of q in S can be reported in $O(c_{d,\epsilon} \log n)$ time.

(ii) More generally, given $\epsilon > 0$, $q \in R^d$, and any k, $1 \leq k \leq n$, a sequence of k $(1+\epsilon)$-approximate nearest neighbors of q in S can be computed in $O((c_{d,\epsilon} + kd) \log n)$ time.

In the case of a single nearest neighbor and for fixed d and ϵ, the space and query times given in Theorem 1 are asymptotically optimal in the algebraic decision tree model of computation. This is because $O(n)$ space and $O(\log n)$ time are required to distinguish between the n possible outcomes in which the query point coincides with one of the data points. We make no claims of optimality for the factor $c_{d,\epsilon}$.

Recently there have been a number of results showing that with significantly more storage, it is possible to improve the dimensional dependencies in query time. Clarkson [Cla94] showed that query time could be reduced to $O((1/\epsilon)^{d/2} \log n)$ with $O((1/\epsilon)^{d/2}(\log \rho)n)$ space, where ρ is the ratio between the furthest-pair and closest-pair interpoint distances. Later Chan [Cha97] showed that the factor of $\log \rho$ could be removed from the space complexity. Kleinberg [Kle97] showed that it is possible to eliminate exponential dependencies on dimension in query time, but with $O(n \log d)^{2d}$ space. Recently, Indyk and Motwani [IM98] and independently Kushilevitz, Ostrovsky and Rabani [KOR98], have announced algorithms that eliminate all exponential dependencies in dimension, yielding a query time $O(d \log^{O(1)}(dn))$ and space $(dn)^{O(1)}$. Here the O-notation hides constant factors depending exponentially on ϵ, but not on dimension.

There are two important practical aspects of Theorem 1. First, space requirements are completely independent of ϵ and are asymptotically optimal for all parameter settings, since dn storage is needed just to store the data points. In applications where n is large and ϵ is small, this is an important consideration. Second, preprocessing is independent of ϵ and the metric, implying that once the data structure has been built, queries can be answered for any error bound ϵ and for any Minkowski metric. In contrast, all the above mentioned methods would require that the data structure be rebuilt if ϵ or the metric changes. In fact, setting $\epsilon = 0$ will cause our algorithm to compute the true nearest neighbor, but we cannot provide bounds on running time, other than a trivial $O(dn \log n)$ time bound needed to search the entire tree by our search algorithm. Unfortunately, exponential factors in query time do imply that our algorithm is not practical for large values of d. However, our empirical evidence in Section 6 shows that the constant factors are much smaller than the bound given in Theorem 1 for the many distributions that we have tested. Our algorithm can provide significant improvements over brute-force search in dimensions as high as 20, with a relatively small average error. There are a number of important applications of nearest neighbor searching in this range of dimensions.

The algorithms for both preprocessing and queries are deterministic and easy to implement. Our data structure is based on a hierarchical decomposition of space, which we call a *balanced box-decomposition (BBD) tree*. This tree has $O(\log n)$ height, and subdivides space into regions of $O(d)$ complexity defined by axis-aligned hyperrectangles that are *fat*, meaning that the ratio between the longest and shortest sides is bounded. This data structure is similar to balanced structures based on box-decomposition [BET93, CK95, Bes95], but there are a few new elements that have been included for the purposes of nearest neighbor searching and practical efficiency. Space is recursively subdivided into a collection of *cells*, each of which is either a d-dimensional rectangle or the set-theoretic difference of two rectangles, one enclosed within the other. Each node of the tree is associated with a cell, and hence it is implicitly associated with the set of data points lying within this cell. Each leaf cell is associated with a single point lying within the bounding rectangle for the cell. The leaves of the tree define a subdivision of space. The tree has $O(n)$ nodes and can be built in $O(dn \log n)$ time.

Here is an intuitive overview of the approximate nearest neighbor query algorithm. Given the query point q, we begin by locating the leaf cell containing the query point in $O(\log n)$ time by a simple descent through the tree. Next, we begin enumerating the leaf cells in increasing order of distance from the query point. We call this *priority search*. When a cell is visited, the distance from q to the point associated with this cell is computed. We keep track of the closest point seen so far. For example, Figure 1(a) shows the cells of such a subdivision. Each cell has been numbered according to its distance from the query point.

Let p denote the closest point seen so far. As soon as the distance from q to the current leaf cell exceeds $dist(q,p)/(1+\epsilon)$ (illustrated by the dotted circle in Figure 1(a)), it follows that the search can be terminated, and p can be reported as an approximate nearest neighbor to q. The reason is that any point located in a subsequently visited cell cannot be close enough to q to violate p's claim to be an approximate nearest neighbor. (In the example shown in the figure, the search terminates just prior to visiting cell 9. In this case p is not the true nearest neighbor, since that point belongs to cell 9, which was never visited.) We will show that, by using an auxiliary heap, priority search can be performed in time $O(d \log n)$ times the number of leaf cells that are visited.

We will also show that the number of cells visited in the search depends on d and ϵ, but not on n. Here is an intuitive explanation (and details will be given in Lemma 5). Consider the last leaf cell to be visited that did not cause the algorithm to terminate. If we let r denote the distance from q to this cell, and let p denote the closest data point encountered so far, then because we do

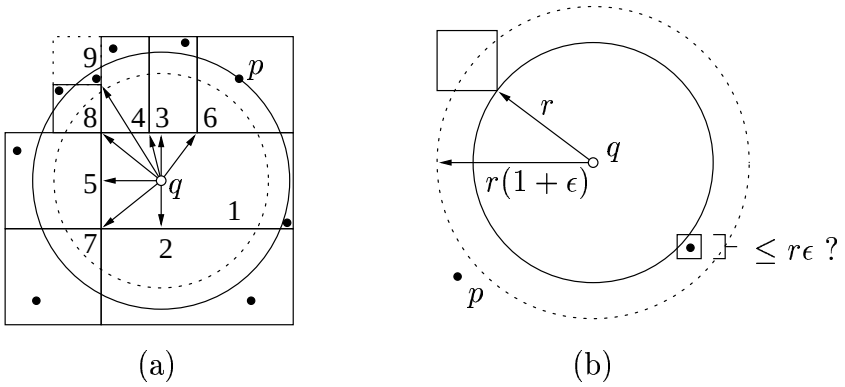

Figure 1: Algorithm overview.

not terminate, we know that the distance from q to p is at least $r(1+\epsilon)$. (See Figure 1(b).) We could not have seen a leaf cell of diameter less than $r\epsilon$ up to now, since the associated data point would necessarily be closer to q than p. This provides a lower bound on the sizes of the leaf cells seen. The fact that cells are fat and a simple packing argument provide an upper bound on the number of cells encountered.

It is an easy matter to extend this algorithm to enumerate data points in "approximately" increasing distance from the query point. In particular we will show that a simple generalization to this search strategy allows us to enumerate a sequence of k approximate nearest neighbors of q in additional $O(kd \log n)$ time. We will also show that, as a consequence of the results of Callahan and Kosaraju [CK95] and Bespamyatnikh [Bes95], the data structure can be generalized to handle point insertions and deletions in $O(\log n)$ time per update.

The rest of the paper is organized as follows. In Section 2 we introduce the BBD-tree data structure, present an algorithm for its construction, and analyze its structure. In Section 3 we establish the essential properties of the BBD-tree which are used for the nearest neighbor algorithm. In Section 4 we present the query algorithm for the nearest neighbor problem, and in Section 5 we present its generalization to enumerating the k approximate nearest neighbors. In Section 6 we present experimental results.

2 The BBD-tree.

In this section we introduce the *balanced box-decomposition tree* or *BBD-tree*, which is the primary data structure used in our algorithm. It is among the general class of geometric data structures based on a hierarchical decomposition of space into d-dimensional rectangles whose sides are orthogonal to the coordinate axes. The main feature of the BBD-tree is that it combines in one data structure two important features that are present in these data structures.

First consider the optimized kd-tree [FBF77]. This data structure recursively subdivides space by a hyperplane that is orthogonal to one of the coordinate axes and which partitions the data points as evenly as possible. As a consequence, as one descends any path in the tree the *cardinality* of points associated with the nodes on this path decreases exponentially. In contrast, consider quadtree-based data structures, which decompose space into regions that are either hypercubes, or generally rectangles whose *aspect ratio* (the ratio of the length of the longest side to the shortest side) is bounded by a constant. These include PR-quadtrees (see Samet [Sam90]), and structures by Clarkson [Cla83], Feder and Greene [FG88], Vaidya [Vai89], Callahan and Kosaraju [CK92],

and Bern [Ber93], among others. An important feature of these data structures is that as one descends any path in these trees, the geometric *size* of the associated regions of space (defined, for example, to be the length of the longest side of the associated rectangle) decreases exponentially. The BBD-tree is based on a spatial decomposition that achieves both exponential cardinality and geometric size reduction as one descends the tree.

The BBD-tree is similar to other balanced structures based on spatial decomposition into rectangles of bounded aspect ratio. In particular, Bern, Eppstein, and Teng [BET93], Schwarz, Smid, and Snoeyink [SSS94], Callahan and Kosaraju [CK95], and Bespamyatnikh [Bes95] have all observed that the unbalanced trees described earlier can be combined with auxiliary balancing data structures, such as centroid decomposition trees [Cha82], dynamic trees [ST83], or topology trees [Fre93] to achieve the desired combination of properties. However, these auxiliary data structures are of considerable complexity. We will show that it is possible to build a single balanced data structure without the need for any complex auxiliary data structures. (This is a major difference between this and the earlier version of this paper [AMN+94].)

The principal difference between the BBD-tree and the other data structures listed above is that each node of the BBD-tree is associated not simply with a d-dimensional rectangle, but generally with the set theoretic difference of two such rectangles, one enclosed within the other. Note, however, that any such region can always be decomposed into at most $2d$ rectangles by simple hyperplane cuts, but the resulting rectangles will not generally have bounded aspect ratio. We show that a BBD-tree for a set of n data points in R^d can be constructed in $O(dn \log n)$ time and has $O(n)$ nodes.

Before describing the construction algorithm, we begin with a few definitions. By a *rectangle* in R^d we mean the d-fold product of closed intervals on the coordinate axes. For $1 \leq i \leq d$, the ith *length* of a rectangle is the length of the ith interval. The *size* of a rectangle is the length of its longest side. We define a *box* to be a rectangle whose aspect ratio (the ratio between the longest and shortest sides) is bounded by some constant, which for concreteness we will assume to be 3.

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

An important concept which restricts the nature of inner boxes is a property called *stickiness*. Consider a cell with outer box b_O and inner box b_I. Intuitively, the box b_I is sticky for b_O if each face is either sufficiently far from or else touching the corresponding face of b_O. To make this precise, consider two closed intervals, $[x_I, y_I] \subseteq [x_O, y_O]$, and let $w = y_I - x_I$ denote the width of the inner interval. We say that $[x_I, y_I]$ is *sticky* for $[x_O, y_O]$ if each of the two distances between the inner interval and the outer interval, $x_I - x_O$ and $y_O - y_I$, is either 0 or at least w. The inner box b_I is *sticky* for the outer box b_O if each of the d intervals of b_I is sticky for the corresponding interval of b_O. (See Figure 2(a).) An equivalent condition for stickiness arises by considering the 3^d regular grid of copies of b_I, centered around b_I. Observe that b_I is sticky for b_O if and only if each rectangle in this grid either lies entirely within b_O or is disjoint from the interior of b_O. (See the lower right box in Figure 2(a).) Throughout, we maintain the property that for all cells the inner box is sticky for the outer box. Stickiness is needed for various technical reasons, which will be discussed later. In particular, it prohibits situations in which a large number of inner boxes are nested one within the next, and all are extremely close to the outer wall of a cell. In situations like this it will not be possible to prove our bounds on the search time.

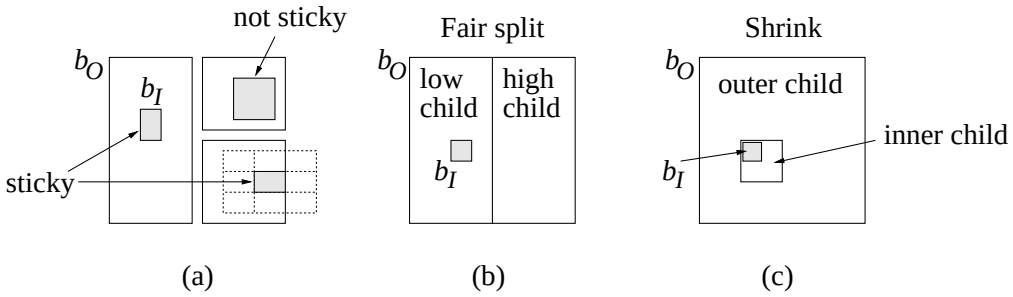

Figure 2: Stickiness, fair splits, and shrinking.

2.1 Overview of the Construction Process

The BBD-tree is constructed through the repeated application of two operations, *fair splits* (or simply *splits*) and *shrinks*. These operations will be described in detail later, but intuitively they represent two different ways of subdividing a cell into two smaller cells, called its *children*. A fair split partitions a cell by an axis-orthogonal hyperplane. The two children are called the *low child* and *high child*, depending on whether the coordinates along the splitting coordinate are less than or greater than the coordinate of the splitting plane. (See Figure 2(b).) A shrink partitions a cell into disjoint subcells, but uses a box (called the *shrinking box*) rather than a hyperplane to do the splitting. It partitions a cell into two children, one lying inside (the *inner child*) and one lying outside (the *outer child*). (See Figure 2(c).) Both operations are performed so that the following invariants hold.

- All boxes satisfy the aspect ratio bound.
- If the parent has an inner box, then this box lies entirely within one of the two children. If the operation is a shrink, then this inner box lies within the inner child of the shrink.
- Inner boxes are sticky for their enclosing outer boxes.

Observe that when only fair splits are performed, it may not generally be possible to guarantee that the points are partitioned evenly. Hence a tree resulting from fair splits alone may not be balanced. (The "fairness" of the split refers to the aspect ratio bound, not to the balance of the point partition.) Intuitively the shrink operation remedies this shortcoming by providing the ability to rapidly zoom into regions where data points are highly clustered.

Note that a split is really a special case of shrink, where the shrinking box has $2d - 1$ sides in common with the outer box. There are two reasons for making the distinction. The first is that splitting will be necessary for technical reasons in maintaining the above invariants. The other reason is largely practical. Determining whether a point lies within a shrinking box requires $2d$ comparisons in general. On the other hand, determining the side of a splitting hyperplane on which a point lies requires only one comparison. For example, in dimension 10, this represents a 20-to-1 savings. We will also see that programming tricks for incrementally updating distances are most efficient when splitting is used.

The BBD-tree is constructed through a judicious combination of fair split and shrink operations. Recall that we are given a set S of n data points in R^d. Let C denote a hypercube containing all the points of S. The root of the BBD-tree is a node whose associated cell is C and whose associated set is the entire set S. The recursive construction algorithm is given a cell and a subset of data points

associated with this cell. Each stage of the algorithm determines how to subdivide the current cell, either through splitting or shrinking, and then partitions the points among the child nodes. This is repeated until the number of associated points is at most one (or more practically is less than some small constant, called the *bucket size*), upon which the node is made a leaf of the tree.

Given a node with more than one data point, we first consider the question of whether we should apply splitting or shrinking. As mentioned before, splitting is preferred because of its simplicity, and the fact that after every d consecutive splits, the geometric size of the associated cell decreases by a constant factor. Splitting cannot guarantee that the point set is partitioned evenly, but we will see that shrinking can provide this. A simple strategy (which we will assume in proving our results) is that splits and shrinks are applied alternately. This will imply that both the geometric size and the number of points associated with each node will decrease exponentially as we descend a constant number of levels in the tree. A more practical approach, which we have used in our implementation, is to perform splits exclusively, as long as the cardinalities of the associated data sets decrease by a constant factor after a constant number of splits. If this condition is violated, then a shrink is performed instead. Our experience has shown that shrinking is only occasionally needed, in particular for data sets that arise from highly clustered distributions, but it can be critical for the efficiency of the search in these cases.

Once it has been determined whether to perform a split or a shrink, the splitting plane or shrinking box is computed, by a method to be described later. For now, let us assume that this can be done in time proportional to the number of points associated with the current node. Once the splitting plane or shrinking box is known, we store this information in the current node, create and link the two children nodes into the tree, and then partition the associated data points among these children. If any data points lie on the boundary of the splitting surface, then these points are distributed among the two children so that the final partition is as even as possible. Finally we recurse on the children.

2.2 Partitioning Points

Before presenting the details on how the splitting plane or shrinking box is computed, we describe how the points are partitioned. We employ a technique for partitioning multidimensional point sets due to Vaidya [Vai89]. We assume that the data points that are associated with the current node are stored in d separate doubly-linked lists, each sorted according to one of the coordinate axes. Actually, the coordinates of each point are stored only once. Consider the list for the ith coordinate. Each entry of this doubly-linked list contains a pointer to the point's coordinate storage, and also a *cross link* to the instance of this same point in the list sorted along coordinate $i+1$ (where indices are taken modulo d). Thus, if a point is deleted from any one list, it can be deleted from all other lists in $O(d)$ time by traversing the cross links. Since each point is associated with exactly one node at any stage of the construction, the total space needed to store all these lists is $O(dn)$. The initial lists containing all the data points are built in $O(dn \log n)$ time by sorting the data points along each of the d coordinates.

To partition the points, we enumerate the points associated with the current node, testing which side of the splitting plane or shrinking box each lies. We label each point accordingly. Then in $O(dn)$ time it is easy to partition each of the d sorted lists into two sorted lists. Since two nodes on the same level are associated with disjoint subsets of S, it follows that the total work to partition the nodes on a given level is $O(dn)$. We will show that the tree has $O(\log n)$ depth. From this it will follow that the total work spent in partitioning over the entire construction algorithm will be $O(dn \log n)$. (The d sorted lists are not needed for the efficiency of this process, but they will be

needed later.)

To complete the description of the construction algorithm, it suffices to describe how the splitting plane and shrinking box are computed and show that this can be done in time linear in the number of points associated with each node. We will present two algorithms for these tasks, the *midpoint algorithm* and the *middle-interval algorithm* (borrowing a term from [Bes95]). The midpoint algorithm is conceptually simpler, but its implementation assumes that nonalgebraic manipulations such as exclusive-or, integer division, and integer logarithm can be performed on the coordinates of the data points. In contrast, the middle-interval algorithm does not make these assumptions, but is somewhat more complex. The midpoint algorithm is a variant of the one described in an earlier version of this paper [AMN+94], and the middle-interval algorithm is a variant of the algorithm given by Callahan and Kosaraju [CK95] and developed independently by Bespamyatnikh [Bes95].

2.3 Midpoint Algorithm

The midpoint algorithm is characterized by restricting the types of rectangles that can arise in the construction. Define a *midpoint box* to be any box that can arise by a recursive application of the following rule, starting from the initial bounding hypercube C.

Midpoint splitting rule: Let b be a midpoint box, and let i be the longest side of b (and among all sides having the same length, i has the smallest coordinate index). Split b into two identical boxes by a hyperplane passing through the center of b and orthogonal to the ith coordinate axis. (See Figure 3(a).)

This can be seen as a binary variant of the standard quadtree splitting rule [Sam90]. We split through the midpoint each time by a cyclically repeating sequence of orthogonal hyperplanes. The midpoint algorithm uses only midpoint boxes in the BBD-tree. It is easy to verify that every midpoint box has an aspect ratio of either 1 or 2. If we assume that C is scaled to the unit hypercube $[0,1]^d$ then the length of each side of a midpoint box is a nonnegative power of $1/2$, and if the ith length is $1/2^k$, then the endpoints of this side are multiples of $1/2^k$. It follows that if b_O and b_I are midpoint boxes, with $b_I \subset b_O$, then b_I is sticky for b_O (since the ith length of b_O is at least as long as that of b_I, and hence is aligned with either the same or a smaller power of $1/2$). Thus we need to take no special care to enforce stickiness. Another nice consequence of using midpoint boxes is that the boxes are contained hierarchically within one another. This implies that inner boxes lie entirely to one side or the other of each fair split and shrink.

To perform a fair split, we simply perform a midpoint split. The shrinking operation is more complicated. Shrinking is performed as part of a global operation called a *centroid shrink*, which will produce up to three new nodes in the tree (two shrinking nodes and one splitting node). Let n_c denote the number of data points associated with the current cell. The goal of a centroid shrink is to decompose the current cell into a constant number of subcells, each containing at most $2n_c/3$ data points. We begin by giving a description of a *simplified approach* to the centroid shrink, which does not quite work, and then we show how to fix it.

We apply a midpoint split to the outer box of this cell, creating two cells. If both of the cells contain no more than $2n_c/3$ data points, then we are done. (And the centroid shrink degenerates to a single split.) Otherwise, we take the cell containing the larger number of points and again apply a midpoint split to it. We repeat this process, always splitting the cell with the majority of points, until first arriving at a cell that contains no more than $2n_c/3$ points. (See Figure 3(b).) The outer

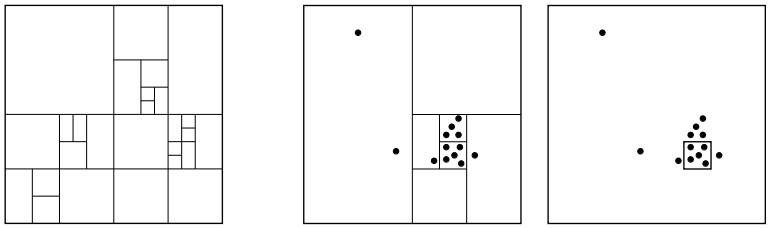

Figure 3: Midpoint construction: Midpoint boxes and centroid shrinking.

box of this cell is the shrinking box for the shrink operation. The intermediate splits used in the creation of this box are simply discarded and generate no new nodes in the BBD-tree. Observe that prior to the last split we had a box with at least $2n_c/3$ data points, and hence the shrinking box contains at least $n_c/3$ points. Thus, there are at most $2n_c/3$ points inside the shrinking box and at most $2n_c/3$ points outside the shrinking box.

There are two problems with this construction.

Problem 1: The number of midpoint splits needed until this procedure terminates cannot generally be bounded by a function of n_c (e.g., when the data points are very tightly clustered).

Problem 2: The resulting shrinking box does not necessarily contain the inner box of the original cell, as required in the shrink operation.

To remedy Problem 1, we need to accelerate the decomposition algorithm when points are tightly clustered. Rather than just repeatedly splitting, we combine two operations, first shrinking to a tight enclosing midpoint box and then splitting this box. From the sorted coordinate lists, we can determine a minimum bounding rectangle (not necessarily a box) for the current subset of data points in $O(d)$ time. Before applying each midpoint split, we first compute the smallest midpoint box that contains this rectangle. We claim that this can be done in $O(d)$ time, assuming a model of computation in which exclusive-or, integer floor, powers of 2, and integer logarithm can be computed on point coordinates. (We omit the proof here, since we will see in the next section that this procedure is not really needed for our results. See Bern, et al. [BET93] for a solution to this problem based on a bit-interleaving technique.) Then we apply the split operation to this minimal enclosing midpoint box. From the minimality of the enclosing midpoint box, it follows that this split will produce a nontrivial partition of the point set. Therefore, after at most $n_c/3 = O(n_c)$ repetitions of this shrink-and-split combination, we will have succeeded in reducing the number of remaining points to at most $2n_c/3$.

To remedy Problem 2, we replace the single stage shrink described in the simple approach with a 3-stage decomposition, which shrinks, then splits, then shrinks. Suppose that we are applying the centroid shrink to a cell that contains an inner box b_I. When we compute the minimum enclosing rectangle for the data points, we make sure that it includes b_I as well. This can be done easily in $O(d)$ time, given the minimum enclosing rectangle for the data points. Now we apply the above iterated shrinking/splitting combination, until (if ever) we first encounter a split that separates b_I from the box containing the majority of the remaining points. Let b denote the box that was just split. (See Figure 4(b).) We create a shrinking node whose shrinking box is b. The outer child contains the points lying outside of b. The inner child consists of a splitting node, with the box

containing b_I on one side, and the box containing the majority of the data points on the other side. Finally, we continue with the centroid shrinking procedure with the child cell that contains the majority of points. Since this cell has no inner box, the above procedure will correctly compute the desired shrinking node. The nodes created are illustrated in Figure 4(c). The final result from the centroid shrink is box c in the lower left. Note that this figure illustrates the most general case. For example, if the first split separates b_I from the majority, then there is no need for the first shrinking node. The (up to) four remaining cells are decomposed recursively. Also note that none of these cells contains more than $2n_c/3$ data points.

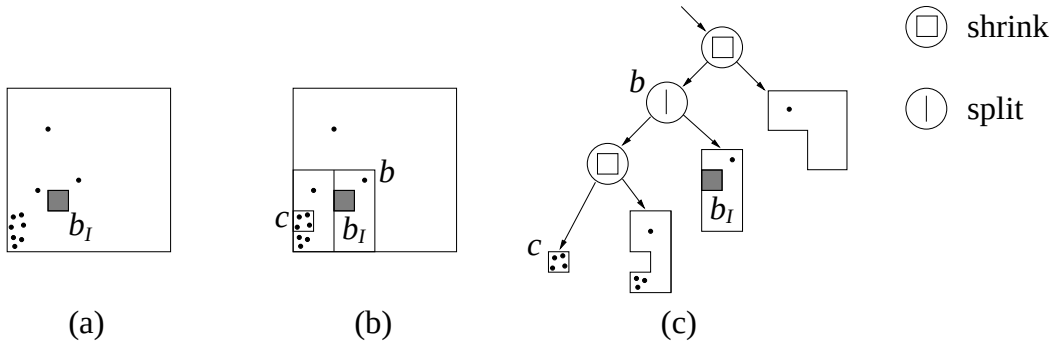

Figure 4: Midpoint construction: Centroid shrinking with an inner box.

Lemma 1 *Given a parent node associated with n_c points, and assuming that the points have been stored in coordinate-sorted lists, each split and centroid shrink can be performed in $O(dn_c)$ time.*

Proof: The centroid shrink is clearly the more complex of the two operations, so we will present its analysis only. We begin by making a copy of the d coordinate-sorted point lists described earlier in $O(dn_c)$ time. Now consider each split used in finding the centroid box. In $O(d)$ time we can compute the minimal enclosing midpoint box, and the splitting plane for this box. Letting k denote the number of points in the box and j denote the number of points on the smaller side of the split, we show that we can eliminate these j points in $O(dj)$ time. Suppose that we are splitting along the ith coordinate. By walking along the ith list, inward from both ends, we can determine which of the two subsets of the partition is smaller in $O(j)$ time. Then we remove the points of this subset from this list, and remove them from the other $d-1$ lists as well in $O(dj)$ time by traversing the cross links. Now the lists contain only the data points from the larger subset of size $k-j$ and are still in sorted order. We pass this list along to the next iteration.

Since (after finding a minimum enclosing box) each split is nontrivial, each such split eliminates from 1 to $k/2$ points from further consideration. Letting $T(k)$ denote the time to complete the processing on a subset of k points, we can see that (ignoring constant factors and $O(dn_c)$ time for initial copying and final point partitioning) the running time is given by the following recurrence.

$$\begin{aligned} T(k) &= 1 && \text{if } k \le 2n_c/3, \\ T(k) &= \max_{1\le j\le k/2}(dj + T(k-j)) && \text{otherwise.} \end{aligned}$$

An easy induction argument shows that $T(n_c) \le dn_c$, and hence the total running time for each operation is $O(dn_c)$. □

In conclusion, we can compute the splitting plane and shrinking box in $O(dn_c)$ time. Since we alternate splits with shrinks, and shrinking reduces the number of points in each cell by a constant

factor, it follows that the resulting tree has height $O(\log n)$. From the arguments made earlier, it follows that the entire tree can be constructed in $O(dn \log n)$ time.

2.4 Middle-interval algorithm

In this section we present the middle-interval algorithm for constructing splitting planes and shrinking boxes. It does not require the stronger model of computation needed by the previous algorithm. This algorithm also has the advantage of offering a greater degree of flexibility in the choice of splitting planes. Our empirical experience has shown that this flexibility can provide significant (constant factor) improvements in space and query time for highly skewed point distributions. This middle-interval algorithm allows the splitting plane to be chosen from a central strip of the current outer box. The need to maintain stickiness and the aspect ratio bound make the choice of splitting plane somewhat more complicated. The algorithm has the same basic structure as the algorithm presented in the previous section. We describe only the important differences.

First we consider how to perform a fair split on a cell. Let b_O denote the outer box of the current cell. If there is no inner box, then we can split b_O by a hyperplane perpendicular to the longest side and passing through the center of b_O. It is easy to see that, in general, any hyperplane perpendicular to the longest side, and splitting this side into any ratio between 1/3 and 2/3 will partition b_O into two boxes, both of which satisfy the aspect ratio bound. (In practice, alternative choices might be worthwhile to produce a more even data point partition, and hence reduce the height of the tree.)

If there is an inner box b_I, then care must be taken that the splitting hyperplane does not intersect the interior of b_I, and that stickiness is not violated after splitting. Consider the projection of b_I onto the longest side of b_O. If this projection fully covers the longest side of b_O, then we consider the second longest side of b_O, and so on until finding one for which b_I does not fully cover this side. One side must exist since $b_I \neq b_O$. Observe that, by stickiness, this projected interval cannot properly contain the central third of this side. If the projection of b_I lies partially within the central third of this side, we select a splitting plane in the central third passing through a face of b_I (see Figure 5(a)). Otherwise the projection of b_I lies entirely within either the first or last third. In this case we split at the further edge of the center strip (see Figure 5(b)).

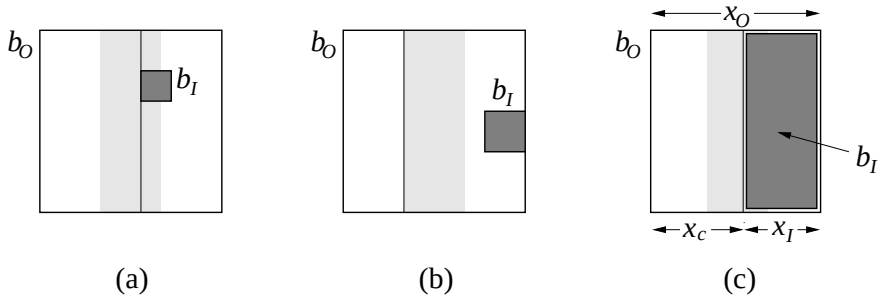

Figure 5: Middle-interval algorithm: Fair split.

It is easy to see that this operation preserves stickiness. We show in the following lemma that the aspect ratio is preserved as well.

Lemma 2 *Given a cell consisting of outer box b_O and inner box b_I satisfying the 3:1 aspect ratio bound, the child boxes produced by the middle-interval split algorithm also satisfy the bound.*

Proof: First observe that the longest side of the two child boxes is not greater than the longest side of b_O. We consider two cases, first where the longest side of b_O is split, and second where some other side of b_O is split. In the first case, if the shortest side of the child is any side other than the one that was split, then clearly the aspect ratio cannot increase after splitting. If the shortest side of the child is the splitting side, then by our construction it is at least one third the length of its parent's longest side, implying that it is at least one third the length of its own longest side.

In the second case, the longest side of b_O was not split. Then by our construction this implies that the projection of b_I along this dimension fully covers this side. It follows that b_I and b_O have the same longest side length, that is, the same size. By hypothesis, b_I satisfies the aspect ratio bound, and so it suffices to show that each side of each child is at least as long as the shortest side of b_I. For concreteness, suppose that the high child contains b_I (as in Figure 5(c)). Clearly the high child satisfies this condition. The low child differs from the high child in only one dimension (namely the dimension that was split). Let x_O, x_I, and x_c denote the lengths of b_O, b_I, and the low child, respectively, along this dimension. We assert that b_I overlaps the middle interval of b_O. If not, then it follows that $x_I < x_O/3 \leq size(b_O)/3 = size(b_I)/3$, contradicting the hypothesis that b_I satisfies the aspect ratio bound. Since b_I overlaps the middle interval, the splitting plane passes through a face of b_I, implying that the distance from b_I to the low side of the low child is x_c. But, since b_I is sticky for b_O, it follows that $x_c \geq x_I$. This completes the proof. □

Computing a centroid shrink is more complicated, but the same approach used in the previous section can still be applied. Recall that the goal is to decompose the current cell into a constant number of cells, such that each contains at most a fraction of 2/3 of the data points. As before, this can be done by repeatedly applying fair splits and recursing on the cell containing the majority of the remaining points, until the number of points falls below 2/3 of the original. Problems 1 and 2, which arose in the previous section, arise here as well. Problem 2 is solved in exactly the same way as before, thus each centroid shrink will generally produce three nodes in the tree, first a shrink to a box containing the old inner box, a split separating the inner box from the majority of points, and a shrink to the new inner box.

To solve Problem 1 we need to eliminate the possibility of performing more than a constant number of splits before succeeding in nontrivially partitioning the remaining points. As before, the idea is to compute a minimal box that encloses both the data points and any inner box that may already be part of the current cell. Achieving both minimality and stickiness is rather difficult, but if r denotes the minimum rectangle (not necessarily a box) which encloses the data points and inner box, then it suffices to construct a box b which contains this rectangle, and whose size is at most a constant factor larger than the size of r. Once such a box is computed, $O(d)$ splits are sufficient to generate a nontrivial partition of r. This in turn implies a nontrivial partition of the point set, or a partition separating the inner box from the majority of points. This box b must also satisfy the stickiness conditions: b is sticky for the current outer box, and the inner box (if it exists) is sticky for b. The construction of such a box is presented in the proof of the next lemma.

Lemma 3 *Given a cell and the minimum bounding rectangle r enclosing both the subset of data points and the inner box of the cell (if there is an inner box), in $O(d)$ time it is possible to construct a box b which is contained within the cell's outer box and which contains r, such that*

(i) the longest side of b is at most a constant factor larger than the longest side of r,

(ii) the cell's inner box (if it exists) is sticky for b, and

(iii) b is sticky for the cell's outer box.

Proof: Let b_O denote the cell's outer box. Recall that the *size* of a rectangle is the length of its longest side. First, observe that if the size of r is within a constant factor of the size of b_O, then we can let $b = b_O$. Otherwise, let us assume that the size of r is at most a factor of $1/36$ of the size of b_O. (We have made no attempt to optimize this constant.) We construct b by applying a series of expansions to r.

First, we consider whether the cell has an inner box. If so, let b_I be this box. By hypothesis, r contains b_I. We expand each side of r so that it encloses the intersection of b_O with the 3^d regular grid of copies of b_I surrounding b_I. (See Figure 6(a).) Note that because b_I is sticky for b_O, this expansion will necessarily lie within b_O. Subsequent expansions of r cannot cause stickiness with respect to b_I to be violated. This may increase the longest side of r by a factor of 3, so the size of r is at most $1/12$ of the size of b_O. Because b_O satisfies the aspect ratio bound, the size of r is at most $1/4$ of the side length of any side of b_O.

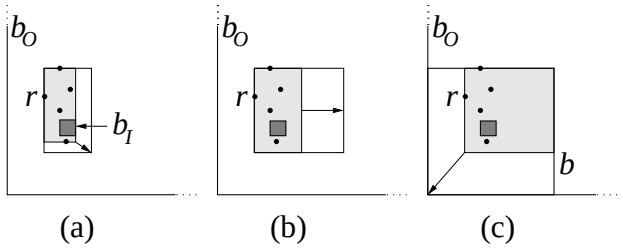

Figure 6: Middle-interval algorithm: Minimal box.

Next, we expand r to form a hypercube. Let l_{max} denote the size of r. Each side of r whose length is less than l_{max} is expanded up to l_{max}. (See Figure 6(b).) Since l_{max} is less than the length of each side of b_O, this expansion can be contained within b_O. This expansion does not increase the length of the longest side of r.

Finally, we consider whether r is sticky for b_O. If it is not, then we expand each of the violating sides of r until it meets the corresponding side of b_O. (See Figure 6(c).) Let b be this expanded rectangle. Since each side of r is not more than $1/4$ of the length of the corresponding side of b_O, it follows that this expansion will at most double the length of any side of r. (In particular, r may be expanded in one direction along each dimension, but not in both directions.) Thus, the longest side of b is at most $2l_{max}$, and its shortest side is at least l_{max}. Thus, b satisfies the aspect ratio bound. This establishes (i). By the construction, b also satisfies properties (ii) and (iii). The size of b is at most 6 times the size of r. Finally, each of the three expansion steps can easily be performed in $O(d)$ time.

□

This lemma solves Problem 1. The centroid shrinking box is computed essentially as it was in the previous section. We repeatedly compute the enclosing box b described above. Then we perform $O(d)$ splits until nontrivially partitioning the point set. (Note that each trivial split can be performed in $O(1)$ time, since no partitioning is needed.) Finally, we recurse on the larger half of the partition. This process is repeated until the number of points decreases by a factor of $2/3$. In spite of the added complexity, the operation generates only three new nodes in the tree. Partitioning of data points is handled exactly as it was in the previous algorithm. Thus, the entire construction can be performed in $O(dn \log n)$ time.

2.5 Final Modifications

This concludes the description of the construction algorithm for the BBD-tree. However, it will be necessary to perform a few final modifications to the tree, before describing the nearest neighbor algorithm. A split or shrink is said to be *trivial* if one of the children contains no data points. It is possible for the tree construction algorithms to generate trivial splits or shrinks (although it can be shown that there can never be more than a constant number of consecutive trivial partitions). It is not hard to see, however, that any contiguous sequence of trivial splits and shrinks can be replaced by a single trivial shrink. We may assume that the data points all lie within the inner box of such a shrinking node, for otherwise we could simply remove this inner box without affecting the subdivision. After constructing the BBD-tree, we replace each such sequence of trivial splits and shrinks by a single trivial shrink. This can be done in $O(n)$ time by a simple tree traversal.

We would like to be able to assume that each leaf contains at least one data point, but this is generally not the case for the leaf nodes resulting from trivial shrinks. We claim that we can associate a data point with each such empty leaf cell by *borrowing* a point from its inner box. Furthermore, we claim that this can be done so that each data point is associated with at most two leaf cells. To see this, consider the following borrowing procedure. Each nontrivial split or shrink node recursively borrows one point from each of its two children, and passes these to its parent. If the parent is a trivial shrink, it uses one of the points for its empty leaf child, and passes the other up the tree. Because there are no two consecutive trivial shrinks or splits, the grandparent must be nontrivial, and so this procedure succeeds in borrowing a different data point for each empty leaf. In summary we have the following characterization of the BBD-tree.

Theorem 2 *Given a set of n data points S in R^d, in $O(dn \log n)$ time we can construct a binary tree having $O(n)$ nodes representing a hierarchical decomposition of R^d into cells (satisfying the stickiness properties given earlier) such that*

(i) The height of the tree is $O(\log n)$ and in general, with every 4 levels of descent in the tree, the number of points associated with the nodes decreases by at least a factor $2/3$.

(ii) The cells have bounded aspect ratio, and with every $4d$ levels of descent in the tree, the sizes of the associated cells decrease by at least a factor of $2/3$.

(iii) Each leaf cell is associated with one data point, which is either contained within the cell, or contained within the inner box of the cell. No data point is associated with more than two leaf cells.

Proof: We assume a construction in which centroid shrinks are alternated with fair splits. Construction time and space and property (iii) follow from the earlier discussion in this section. To see (i), observe that because each centroid shrink introduces three new levels into the tree, and we alternate this with fair splits, it follows that with each four levels the number of points decreases by at least a factor of $2/3$. For (ii), note that to decrease the size of a cell, we must decrease its size along each of d dimensions. Since fair splits are performed at least at every fourth level of the tree, and each such split decreases the longest side by at least a factor of $2/3$, it follows that after at most d splits (that is, at most $4d$ levels of the tree) the size decreases by this factor. □

The BBD-tree which results from this construction is quite similar to the tree described in the earlier version of this paper [AMN+94]. The main differences are that centroid shrinking has been

incorporated into the tree (through the use of the centroid shrink operation), and the cells associated with internal nodes of the tree are of bounded complexity. These properties significantly simplify the implementation of the data structure. The size reduction property mentioned in Theorem 2(ii) is not used in this paper, but it is important in other applications of BBD-trees for geometric approximation problems [AM95, MNSW95].

Although we do not know how to maintain the BBD-tree directly under point insertion and deletion, by using an auxiliary data structure (either a *topology tree* [Fre93] or a *dynamic tree* [ST83]) to represent the unbalanced box-decomposition tree, it is possible to support data point insertions and deletions in $O(\log n)$ time each. See either Callahan and Kosaraju [CK95] or Bespamyatnikh [Bes95] for details. A somewhat more practical approach to insertion and deletion would be to achieve $O(\log n)$ amortized time for insertion and deletion by rebuilding unbalanced subtrees, using the same ideas as *scapegoat trees* [GR93]. The key fact is that given an arbitrarily unbalanced subtree of a box-decomposition tree, it is possible to replace it with a balanced subtree (representing the same underlying spatial subdivision) in time linear in the size of the subtree. For example, this can be done by building a topology tree for the subtree [Fre85].

3 Essential Properties

Before describing the nearest neighbor algorithm, we enumerate some important properties of the BBD-tree, which will be relevant to nearest neighbor searching. These will be justified later. Recall that each cell is either a rectangle, or the difference of two rectangles, one contained within the other. Recall that the leaf cells of the BBD-tree form a subdivision of space. The cells of this subdivision satisfy the following properties.

(a) *Bounded occupancy:* Each cell contains up to some constant number of data points (possibly zero). Points that lie on the boundary between two or more cells are assigned to one of the cells.

(b) *Existence of a nearby data point:* If a cell contains one or more data points, then these points are associated with the cell. Otherwise, a data point lying within the cell's outer box is associated with the cell. This is done in such a way that each data point is associated with $O(1)$ different cells.

(c) *Point location:* Given a point q in R^d, a cell of the subdivision containing q can be determined in $O(d \log n)$ time.

(d) *Packing constraint:* The number of cells of size at least s that intersect an open ball of radius $r > 0$ is bounded above by a function of r/s and d, but independent of n. (By *ball* we mean the locus of points that are within distance r of some point in R^d according to the chosen Minkowski distance metric.)

(e) *Distance enumeration of cells:* Define the *distance* between a point q and a cell to be the closest distance between q and any part of the cell. Given q, the cells of the subdivision can be enumerated in order of increasing distance from q. The m nearest cells can be enumerated in $O(md \log n)$ time.

Properties (a) and (b) are immediate consequences of our construction. In particular, each leaf cell contains at most one point, and each point is associated with at most two different cells.

Property (c) follows from a simple descent through the tree. The following lemma establishes property (d), and (e) will be established later. The proof of the lemma makes critical use of the aspect ratio bound and the stickiness property introduced earlier.

Lemma 4 (Packing Constraint) *Given a BBD-tree for a set of data points in R^d, the number of leaf cells of size at least $s > 0$ that intersect a Minkowski L_m ball of radius r is at most $\lceil 1 + 6r/s \rceil^d$.*

Proof: From the 3:1 aspect ratio bound, the smallest side length of a box of size s is at least $s/3$. We say that a set of boxes are *disjoint* if their interiors are pairwise disjoint. We first show that the maximum number of disjoint boxes of side length at least $s/3$ that can overlap any Minkowski ball of radius r is $\lceil 1 + 6r/s \rceil^d$. For any m, an L_m Minkowski ball of radius r can be enclosed in an axis-aligned hypercube of side length $2r$. (The tightest fit is realized in the L_∞ case, where the ball and the hypercube are equal). The densest packing of axis-aligned rectangles of side length at least $s/3$ is realized by a regular grid of cubes of side length exactly $s/3$. Since an interval of length $2r$ can intersect at most $\lceil 1 + 6r/s \rceil$ intervals of length $s/3$, it follows that the number of grid cubes overlapping the cube of side length $2r$ is at most $\lceil 1 + 6r/s \rceil^d$. Therefore this is an upper bound on the number of boxes of side length s that can overlap any Minkowski ball of radius r.

The above argument cannot be applied immediately to the outer boxes of the leaf cells, because they are not disjoint from the leaves contained in their inner boxes. To complete the proof, we show that we can replace any set of leaf cells each of size at least s that overlap the Minkowski ball with an equal number of disjoint boxes (which are not necessarily part of the spatial subdivision) each of size at least s that also overlap the ball. Then we apply the above argument to these disjoint boxes.

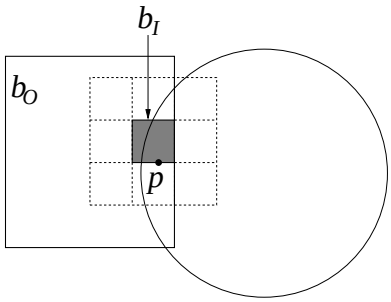

Figure 7: Packing constraint.

For each leaf cell of size at least s that either has no inner box, has an inner box of size less than s, or has an inner box that does not overlap the ball, we take the outer box of this cell to be in the set. In these cases, the inner box cannot contribute a leaf to the set of overlapping cells.

On the other hand, consider a leaf cell c, formed as the difference of an outer box b_O and inner box b_I, such that the size of b_I is at least s, and both b_I and c overlap the ball. Since b_O has at most one inner box, and by convexity of boxes and balls, it follows that there is a point p on the boundary between c and b_I that lies within the ball. Let p denote such a point. (See Figure 7.) Any neighborhood about p intersects the interiors of both c and b_I. By stickiness, we know that the $3^d - 1$ congruent copies b_I, surrounding b_I either lie entirely within b_O or their interiors are disjoint from b_O. Clearly there must be one such box containing p on its boundary, and this box is contained within b_O. (In Figure 7 this is the box lying immediately below p). We take this box to replace c in the set. This box is disjoint from b_I, its size is equal to the size of b_I, and it overlaps the ball. Because leaf cells have disjoint interiors, and each has only one inner box, it follows that

the replacement box will be disjoint from all other replacement boxes. Now, applying the above argument to the disjoint replacement boxes completes the proof. □

The last remaining property to consider is (e), the enumeration of boxes in increasing order of distance from some point q (which we will assume to be the query point). The method is a simple variant of the *priority search* technique used for kd-trees by Arya and Mount [AM93b]. We recall the method here. The algorithm maintains a priority queue of nodes of the BBD-tree, where the priority of a node is inversely related to the distance between the query point and the cell corresponding to the node. Observe that because each cell has complexity $O(d)$, we can compute this distance in $O(d)$ time.

Initially, we insert the root of the BBD-tree into the priority queue. Then we repeatedly carry out the following procedure. First, we extract the node v with the highest priority from the queue, that is, the node closest to the query point. (This is v_1 in Figure 8.) Then we descend this subtree to visit the leaf node closest to the query point. Since each cell consists of the difference of two d-dimensional rectangles, we can determine which child's cell is closer to the query point in $O(d)$ time. We simply recurse down the path of closer children until reaching the desired leaf.

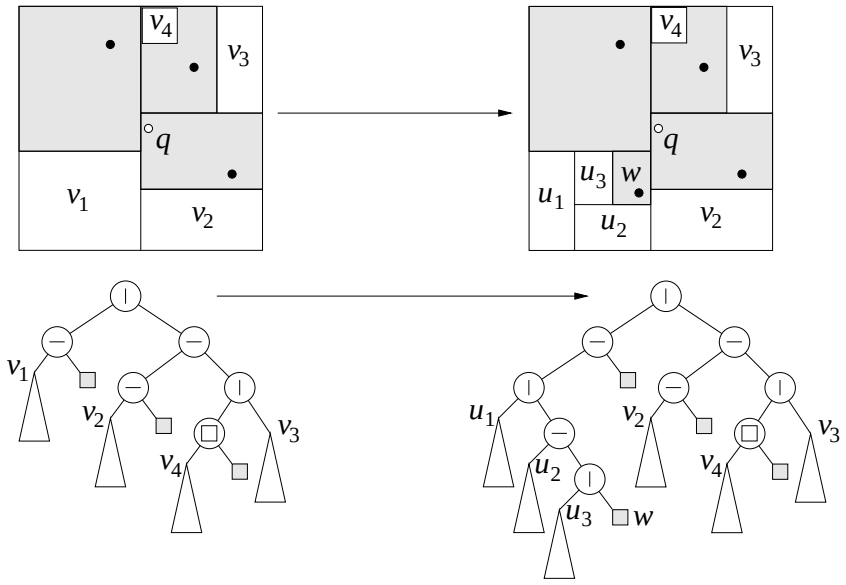

Figure 8: Priority Search.

As we descend the path to this leaf, for each node u that is visited, we compute the distance to the cell associated with u's sibling and then insert this sibling into the priority queue. For example, in Figure 8, subtrees v_1 through v_4 are initially in the queue. We select the closest, v_1, and descend the tree to leaf w, enqueuing the siblings u_1, u_2, and u_3 along the way. The straightforward proof of correctness relies on the invariant that the set of leaves descended from the nodes stored in the priority queue forms a subdivision of the set of all unvisited leaves. This is proved by Arya and Mount [AM93b].

Each node of the tree is visited, and hence enqueued, at most once. Since there are at most n nodes in the heap, we can extract the minimum in $O(\log n)$ time. Each step of the tree descent can be processed in $O(d)$ time (the time to compute the distances from the query point to the child cells) plus the time to insert the sibling in the priority queue. If we assume the use of a Fibonacci heap [FT87] for the priority queue, the amortized time for each insertion is $O(1)$. Since

the BBD-tree has $O(\log n)$ height, and the processing of each internal node takes $O(d)$ time, the next leaf in the priority search can be determined in $O(d \log n)$ time. Thus, the time needed to enumerate the nearest m cells to the query point is $O(md \log n)$. Thus property (e) is established.

Before implementing this data structure as stated, there are a number of practical compromises that are worth mentioning. First, we have observed that the size of the priority queue is typically small enough that it suffices to use a standard binary heap (see, e.g., [CLR90]), rather than the somewhat more sophisticated Fibonacci heap. It is also worth observing that splitting nodes can be processed quite a bit more efficiently than shrinking nodes. Each shrinking node requires $O(d)$ processing time, to determine whether the query point lies within the inner box, or to determine the distance from the query point to the inner box. However, it is possible to show that splitting nodes containing no inner box can be processed in time independent of dimension. It takes only one arithmetic comparison to determine on which side of the splitting plane the query point lies. Furthermore, with any Minkowski metric, it is possible to incrementally update the distance from the parent box to each of its children when a split is performed. The construction, called *incremental distance computation* is described in Arya and Mount [AM93b]. Intuitively it is based on the observation that for any Minkowski metric, it suffices to maintain the sum of the appropriate powers of the coordinate differences between the query point and the nearest point of the outer box. When a split is performed, the closer child is at the same distance as the parent, and the further child's distance differs only in the contribution of the single coordinate along the splitting dimension. The resulting improvement in running time can be of significant value in higher dimensions. This is another reason that shrinking should be performed sparingly, and only when it is needed to guarantee balance in the BBD-tree.

4 Approximate Nearest Neighbor Queries.

In this section we show how to answer approximate nearest neighbor queries, assuming any data structure satisfying properties (a)–(e) of the previous section. Let q be the query point in R^d. Recall that the output of our algorithm is a data point p whose distance from q is at most a factor of $(1+\epsilon)$ greater than the true nearest neighbor distance.

We begin by applying the point location algorithm to determine the cell containing the query point q. Next, we enumerate the leaf cells of the subdivision in increasing order of distance from q. Recall from (a) and (b) that each leaf cell is associated with at least one data point that is contained within the outer box of the cell. As each cell is visited, we process it by computing the distance from q to these data points and maintaining the closest point encountered so far. Let p denote this point. The search terminates if the distance r from the current cell to q exceeds $dist(q,p)/(1+\epsilon)$. The reason is that no subsequent point to be encountered can be closer to q than $dist(q,p)/(1+\epsilon)$, and hence p is a $(1+\epsilon)$-approximate nearest neighbor.

From (e) it follows that we can enumerate the m nearest cells to q in $O(md \log n)$ time. To establish the total query time, we apply (d) to bound the number of cells visited.

Lemma 5 *The number of leaf cells visited by the nearest neighbor algorithm is at most* $C_{d,\epsilon} \leq \lceil 1 + 6d/\epsilon \rceil^d$ *for any Minkowski metric.*

Proof: Let r denote the distance from the query point to the last leaf cell that did *not* cause the algorithm to terminate. We know that all cells that have been encountered so far are within

distance r from the query point. If p is the closest data point encountered so far, then because we did not terminate we have

$$r(1+\epsilon) \leq dist(q,p).$$

We claim that no cell seen so far can be of size less than $r\epsilon/d$. Suppose that such a cell was visited. This cell is within distance r of q, and hence overlaps a ball of radius r centered at q. The diameter of this cell in any Minkowski metric is at most d times its longest side length (in general, $d^{1/m}$ times the longest side in the L_m metric), and hence is less than $r\epsilon$. Since the cell is associated with a data point lying within the outer box of the cell, the search must have encountered a data point at distance less than $r + r\epsilon = r(1+\epsilon)$ from q. However, this contradicts the hypothesis that p is the closest point seen so far.

Thus the number of cells visited up until termination is bounded by the number of cells of size at least $r\epsilon/d$ that can overlap a ball of radius r. From property (d) we know that the number of such cells is a function of ϵ and d. Using the bounds derived in Lemma 4, the number of cells is at most $\lceil 1 + 6d/\epsilon \rceil^d$. □

By combining the results of this and previous sections, we have established Theorem 1(i). The extra factor of d differentiating $c_{d,\epsilon}$ in the theorem and $C_{d,\epsilon}$ in the lemma above is due to the $O(d)$ processing time to compute the distance from the query point to each visited node in the tree.

5 Approximate k-Nearest Neighbors.

In this section we show how to generalize the approximate nearest neighbor procedure to the problem of computing approximations to the k nearest neighbors of a query point. Recall that a point p is a $(1+\epsilon)$-approximate jth nearest neighbor to a point q if its distance from q is a factor of at most $(1+\epsilon)$ times the distance to q's true jth nearest neighbor. An answer to the approximate k-nearest neighbors query is a sequence of k distinct data points $p_1, p_2, \ldots, p_k$, such that p_j is a $(1+\epsilon)$-approximation to the j-th nearest neighbor of q, for $1 \leq j \leq k$.

The algorithm is a simple generalization of the single nearest neighbor algorithm. We locate the leaf cell containing the query point, and then enumerate cells in increasing order of distance from q. We maintain the k closest data points to q encountered in the search, say, by storing them in the balanced binary search tree sorted by distance. Let r_k denote the distance to the k-th closest point so far ($r_k = \infty$ if fewer than k distinct points have been seen so far). The search terminates as soon as the distance from the current cell to q exceeds $r_k/(1+\epsilon)$. The reason is that no subsequently visited data point can be closer to q than $r_k/(1+\epsilon)$, and hence the data point at distance r_k is an approximate kth nearest neighbor. There are at least $k-1$ data points that are closer to the query point. It is easy to verify that the sorted sequence of k data points seen so far is a solution to the approximate k-nearest neighbors query. The running time is analyzed below.

Lemma 6 *Recalling $C_{d,\epsilon}$ from Lemma 5, this algorithm visits at most $2k + C_{d,\epsilon}$ leaf cells.*

Proof: To bound the number of leaf cells visited by the algorithm, recall from property (b) that each point is associated with at most two cells. Thus, the k data points reported by the search were contributed by at most $2k$ leaf cells that were visited in the search. We claim that the algorithm encounters at most $C_{d,\epsilon}$ other *noncontributing* leaf cells.

The argument is a straightforward generalization of the one used in Lemma 5. Consider the set of visited leaf cells that did not contribute a point to the final answer. Let r denote the distance to

the last cell of this set that did *not* cause termination. Let p be the kth closest point encountered so far. As in Lemma 5, we have $r(1+\epsilon) \leq dist(q,p)$, and so none of the noncontributing cells seen so far can be of size less than $r\epsilon/d$, or else they would have contributed a point that is closer than p. The final result follows by applying Lemma 4. □

To complete the proof, we recall that the algorithm spends $O(d \log n)$ time to process each leaf cell, and in time $O(\log k) \leq O(\log n)$ we determine whether the point is among the k nearest points encountered so far, and add it to the set if it is. Combining this with the earlier remarks of this section establishes Theorem 1(ii).

6 Experimental Results.

In order to establish the practical value of our algorithms, we implemented them and ran a number of experiments on a number of different data sizes and with point sets sampled from a number of different distributions.

Our implementation differed slightly from the description of the previous sections. First, in preprocessing we did not perform the partitioning using the asymptotically efficient method described in Section 2, of storing the points sorted along each of the d dimensions. Instead we opted for the much simpler technique of applying a standard partitioning algorithm as used in QuickSort (see [CLR90]). This does not affect the structure of the resulting tree, but if splits are very unbalanced then the preprocessing may take longer than $O(dn \log n)$ time. On the other hand, we save a factor of d with each invocation, since only one coordinate is accessed with each partition. Second, we did not use the rather sophisticated algorithms for accelerating the shrinking operation. We just performed repeated splits. We observed no unusually high preprocessing times for the data sets that were tested.

We mentioned earlier that splitting is generally preferred to shrinking because of the smaller factors involved. However, splitting without shrinking may result in trees of size greater than $O(n)$ and height greater than $O(\log n)$. In our implementation we performed shrinking only if a sequence of $d/2$ consecutive splits failed to reduce the fraction of points by at least one half. For most of the distributions that we tested, no shrinking nodes were generated. Even for the highly clustered distributions, a relatively small fraction of shrinking was observed (ranging from 5–20% of the total nodes in the tree). In part, this explains why simple data structures such as kd-trees perform well for most point distributions.

As in [AM93b], incremental distance calculation (described in Section 3) was used to speed up distance calculations for each node. We experimented with two schemes for selecting splitting planes. One was the midpoint-split rule described in Section 2.3 and the other was a variant of the middle-interval rule described in Section 2.4. The latter rule, called the *fair-split rule*, was inspired by the term introduced in [CK92]. Given a box, we first determine the sides that can be split without violating the 3:1 aspect ratio bound. Given a subset of the data points, define the *spread* of these points along some dimension to be the difference between the maximum and minimum coordinates in this dimension. Among the sides that can be split, select the dimension along which the points have maximum spread, and then split along this dimension. The splitting hyperplane is orthogonal to this dimension and is positioned so the points are most evenly distributed on either side, subject to the aspect ratio bound.

We ran experiments on these two data structures, and for additional comparison we also implemented an optimized kd-tree [FBF77]. The cut planes were placed at the median, orthogonal to the

coordinate axis having maximum spread. Although the kd-tree is guaranteed to be of logarithmic depth, there is no guaranteed bound on the aspect ratios of the resulting cells (and indeed ratios in the range from 10:1 to 20:1 and even higher were not uncommon). We know of no prior work suggesting the use of a kd-tree for approximate nearest neighbor queries, but the same termination condition given in Section 4 can be applied here. Unlike the box-decomposition tree, we cannot prove upper bounds on the execution time of query processing. Given the similarity to our own data structure, one would expect that running times would be similar for typical point distributions, and our experiments bear this out.

Our experience showed that adjusting the bucket size, that is, the maximum number of points allowed for each leaf cell, affects the search time. For the more flexible kd-tree and the fair-split rule, we selected a bucket size of 5, but found that for the more restricted midpoint-split rule, a bucket size of 8 produced somewhat better results.

The experiments were run on a Sun Sparc 20 running Solaris. Each experiment consisted of 100,000 data points in dimension 16 and the averages were computed over 1,000 query points. More query points were taken when measuring CPU times, due to greater variations in CPU time caused by varying system loads. For each query we computed the nearest neighbor in the L_2 metric. Except where noted, query points and data points were taken from the same distribution.

Typical preprocessing times ranged from 20 to 100 CPU seconds. The higher running times were most evident with highly clustered data sets and when using the midpoint-split rule. This is because shrinking was needed the most in these cases. In contrast, the optimized kd-tree, whose running time is independent of the data distribution, had preprocessing times uniformly around 20 CPU seconds.

6.1 Distributions Tested

The distributions that we tested are listed below. The correlated Gaussian and correlated Laplacian point distributions were chosen to model data from applications in speech processing. These two point distributions were formed by grouping the output of autoregressive sources into vectors of length d. An autoregressive source uses the following recurrence to generate successive outputs:

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

where W_n is a sequence of zero mean independent, identically distributed random variables. The correlation coefficient 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 [FM85] 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 $[0, 1]$.

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.

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

Clustered Segments: Eight orthogonal 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 plus a Gaussian error with standard deviation of 0.001.

For the clustered segments distribution, five trials were run, with newly generated cluster centers for each trial, and each involving 200 query points. Query points were sampled from a uniform distribution. We show results only for the uniform distribution and two extreme cases, the correlated Laplacian and clustered segments distributions. The results for other distributions generally varied between the uniform case and the correlated Laplacian.

6.2 Query time

For each experiment, we recorded a number of different statistics. We will present only a subset of these statistics, starting with query time. We measured both the average CPU time and the average number of floating point operations for each query. Floating point operations, called *floats*, are any arithmetic operation involving point coordinates or distances. We felt that this provides a reasonable machine-independent measure of the algorithm's running time. A comparison of CPU times and floating operations shows relatively good agreement. We ran experiments for values of ϵ ranging from 0 (exact nearest neighbor) up to 10, in increments of 0.1. The results for the uniform, correlated Laplacian, and clustered segments distributions are shown in Figures 9 through 11. Note that the y-axis is on a logarithmic scale in all cases.

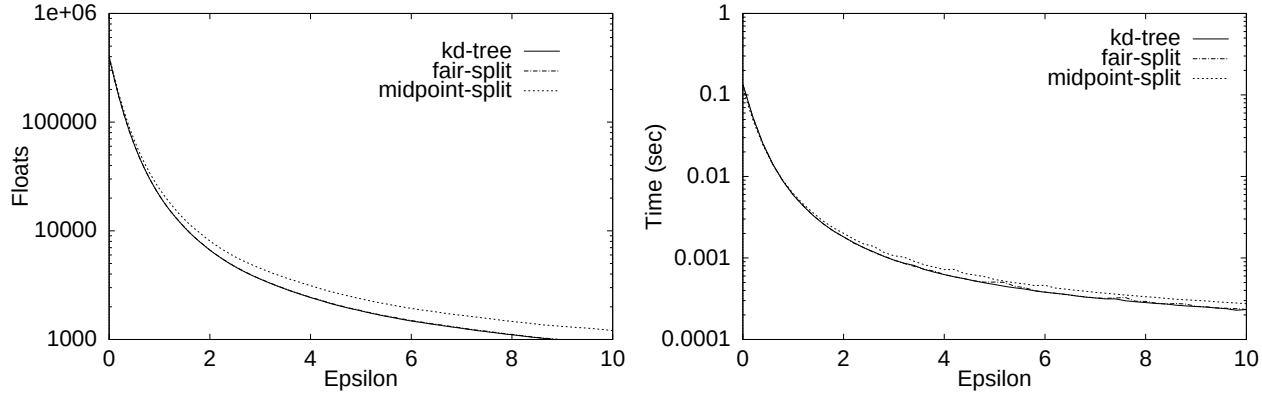

Figure 9: (a) Floating point operations and (b) CPU time versus ϵ for the uniform distribution.

The empirical running times on most of the distributions suggest that there is little or no significant practical advantage to using the BBD-tree over the kd-tree. Indeed, we feel that a kd-tree,

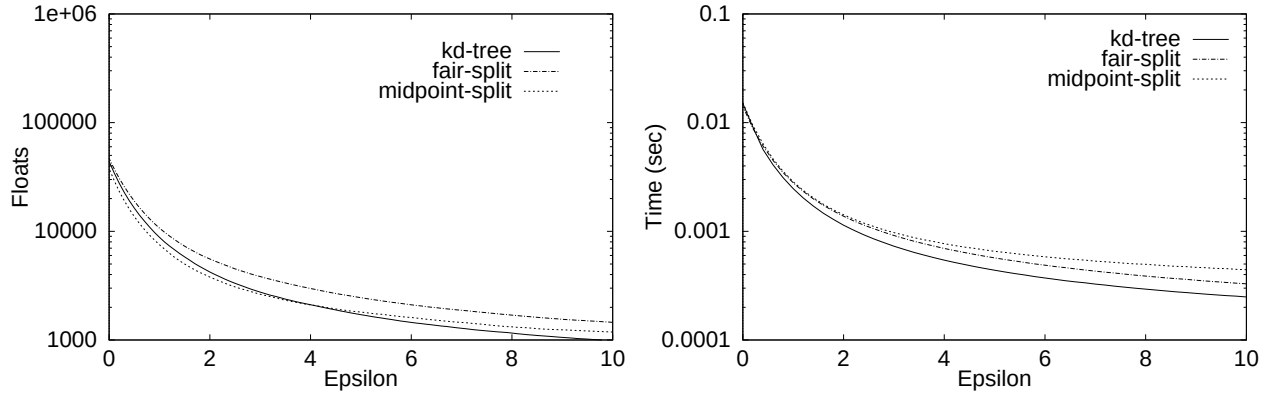

Figure 10: (a) Floating point operations and (b) CPU time versus ϵ for the correlated Laplacian distribution.

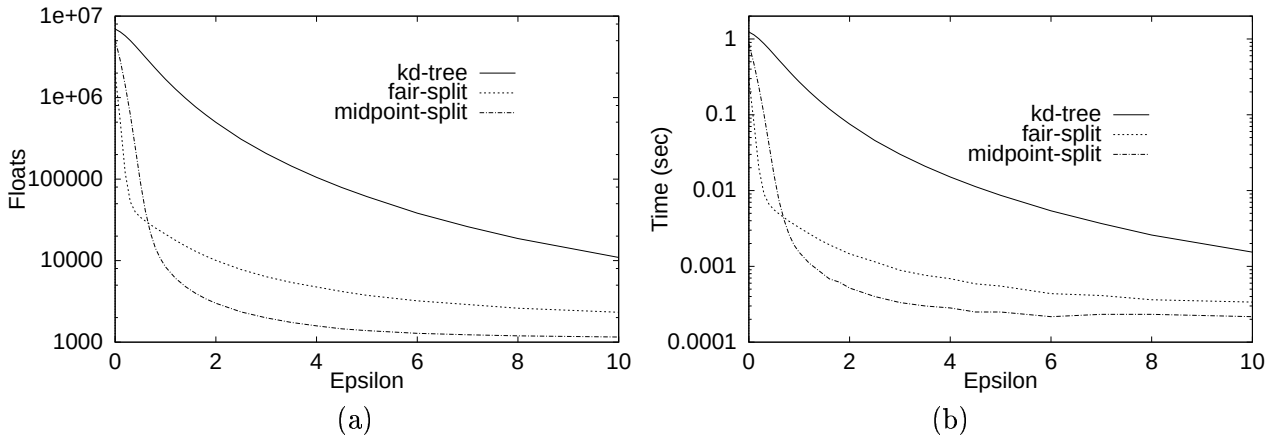

Figure 11: (a) Floating point operations and (b) CPU time versus ϵ for the clustered segments distribution.

enhanced with many of the improvements described in this paper (allowing approximation errors, incremental distance calculations, and priority search) is a very good data structure for nearest neighbor searching on most data sets. However, it can perform very badly in some circumstances, especially when the data distribution is clustered in low-dimensional subspaces, as in the clustered segments distribution. Low-dimensional clustering is not uncommon in practice. An inspection of some of the other program statistics (not shown here) explains why. For this distribution, the kd-tree produced a large number of cells with very high aspect ratios. Because the optimized kd-tree cuts along the dimension of greatest spread, it can produce cells that are very skinny along the dimensions in which the data are well distributed, and very long in the remaining dimensions. These skinny cells violate the packing constraint, which is critical to our analysis. If the query point distribution differs from the data point distribution, then many such skinny cells may be visited by the search. This is why uniformly distributed query points were chosen.

In contrast, we could have forced bounded aspect ratios by using the midpoint splitting rule, but by not allowing shrinking. The result is a sort of binary form of a quadtree. For highly clustered distributions, like clustered segments, this results in trees that are at least an order of magnitude larger than the BBD-tree in both size and depth.

Both variants of BBD-trees took advantage of shrinking to produce reasonably small trees with cells of bounded aspect ratio. As shown in Figure 11, the running times are significantly better than those for the kd-tree for this distribution.

6.3 Average Distance Error

Another issue involves the actual performance of the algorithm with respect to distance errors. The user supplies an upper bound ϵ on the allowable distance error, but the data structure may find points that are closer.

We computed the true nearest neighbor off-line and then computed the actual relative error, namely the ratio between the distance to the point reported by the algorithm and the true nearest neighbor minus 1. The resulting quantity averaged over all query points is called the *average relative error* (or simply *average error*). This is shown in Figure 12 for the uniform and correlated Laplacian distributions. Again, most of the other distributions showed a similar behavior. The results show that for even very large values of ϵ, the average error committed is typically at least an order of magnitude smaller. Although we have no theoretical justification for this phenomenon, this better average-case performance may be of interest in applications where average error over a large number of queries is of interest, and suggests an interesting topic for future study.

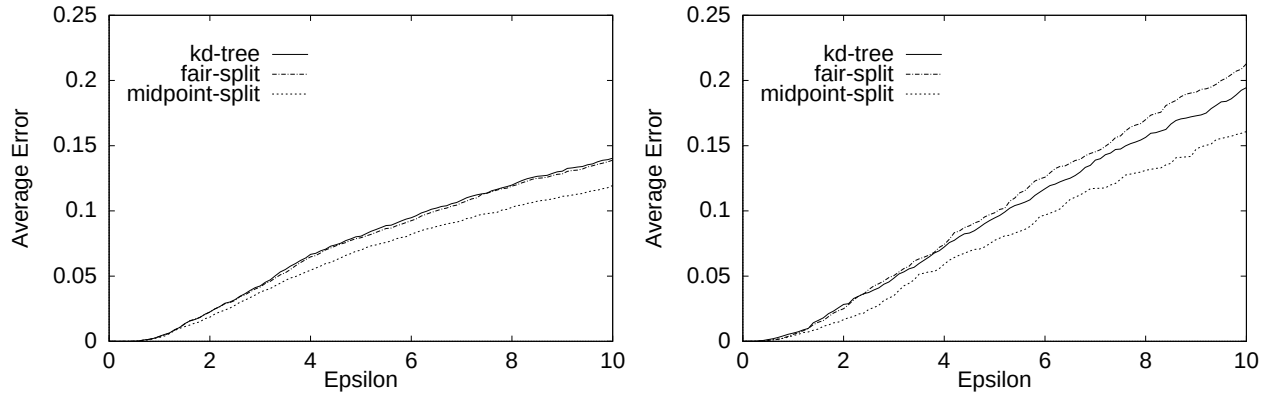

Figure 12: Average error for the (a) uniform and (b) correlated Laplacian distribution versus ϵ.

A related statistic is how often the algorithm succeeds in finding the true nearest neighbor as a function of ϵ. We found that the algorithm manages to locate the true nearest neighbor in a surprisingly large number of instances, even with relative large values of ϵ. To show this, we plotted the fraction of instances in which the algorithm fails to return the true nearest neighbor for these distributions. Results are shown in Figure 13.

6.4 Dependence on Dimension and ϵ

Another question involves the constant factors which depend on the dimension and ϵ. In Lemma 5 we define $C_{d,\epsilon} = \lceil 1 + 6d/\epsilon \rceil^d$, and argue that in all Minkowski metrics, the nearest neighbor can be reported in $O(C_{d,\epsilon} d \log n)$ time. The factor $C_{d,\epsilon}$ bounds the number of leaf cells visited by the algorithm. However, this factor is a crude worst-case bound, which ignores a number of important practical issues. To get a more accurate sense of what sort of factors could be expected in practice,

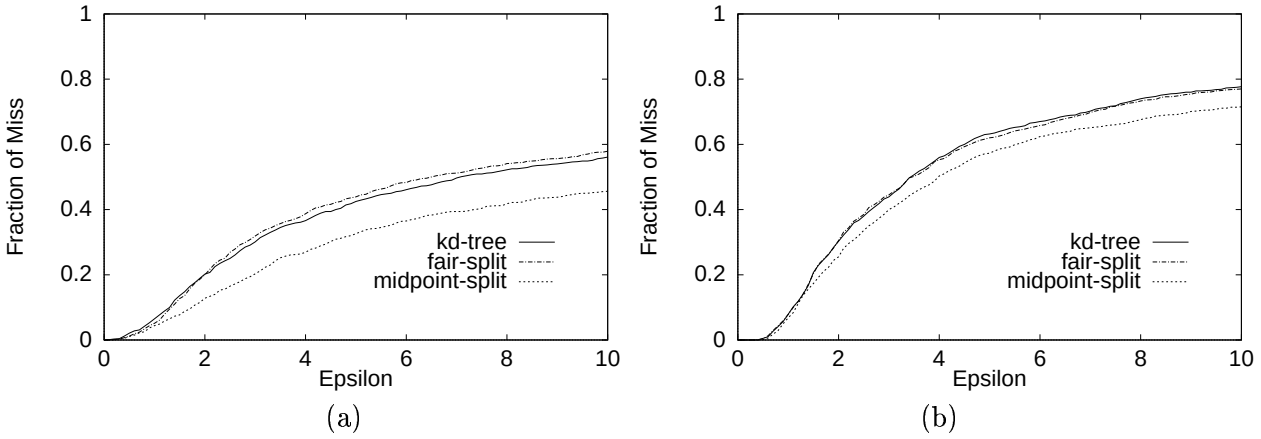

Figure 13: Fraction of nearest neighbors missed for the (a) uniform and (b) correlated Laplacian distributions versus ϵ.

we ran an experiment to measure how the number of cells visited by the algorithm varies as a function of d and ϵ. We also sought an analytical explanation of these results.

We chose a relatively well-behaved case to consider for these experiments, namely uniformly distributed points in a unit hypercube, and the L_∞ metric. Because of the negligible differences in the various data structures for uniformly distributed data (as evidenced by Figure 9 above), we ran experiments only for the kd-tree using a bucket size of 1. We considered dimensions varying from 1 to 16, and values of ϵ varying from 0 to 10. We considered data sets of size 100,000, and for each data set averaged results over 1000 queries.

A plot of the relationship between the logarithm (base 10) of the number of leaf cells visited versus ϵ and dimension is shown in Figure 14(a). Indeed, the figure shows that the number of cells is significantly smaller than the huge values predicted by the above formula. For example, for $\epsilon = 1$ and dimension 16, the formula provides the unusable bound of 10^{32}, whereas the plot shows that the number of cells is roughly 100 for this distribution.

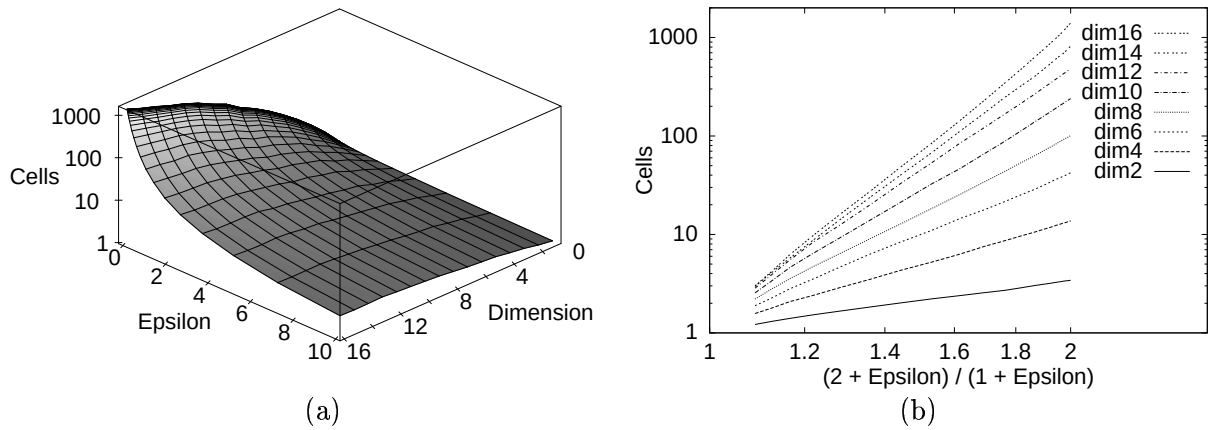

Figure 14: Number of cells visited versus ϵ and dimension.

We can provide an informal analytical justification for these empirical results. We follow the general structure of the analysis by Friedman, Bentley and Finkel [FBF77]. For large uniformly distributed data sets, it is reasonable to model a kd-tree's decomposition of a unit hypercube as a

regular grid of n hypercubes where each hypercube has side length of roughly $a = 1/n^{1/d}$. Ignoring boundary effects, the expected side length of the L_∞ nearest neighbor ball for a random query point is also $1/n^{1/d}$. For $\epsilon > 0$, our algorithm will need to visit any leaf cell that overlaps a shrunken nearest neighbor ball whose side length is $b = a/(1+\epsilon)$. It is easy to see that the expected number of intervals of width a that are overlapped by a randomly placed interval of width b is $(1 + b/a)$. It follows that the number of grid cubes of width a that are overlapped by a randomly placed cube of width b is

$$\left(1 + \frac{b}{a}\right)^d = \left(1 + \frac{1}{1+\epsilon}\right)^d = \left(\frac{2+\epsilon}{1+\epsilon}\right)^d.$$

From this it follows that for any fixed dimension, a linear relationship is to be expected between the logarithm of the number of cells and the logarithm of $(2+\epsilon)/(1+\epsilon)$. This relationship is evidenced in Figure 14(b). (Note that both axes are on a logarithmic scale.) Boundary effects probably play a role since the empirically observed values are somewhat smaller than predicted by the formula [AMN95].

6.5 Summary of Experiments

A number of conclusions can be drawn from these experiments. First, in moderate dimensions, significant savings in running time can be achieved by computing approximate nearest neighbors. For the $\epsilon = 3$ cases, improvements in running time on the order of factors of 10 to 50 over the exact case were common. For clustered data sets, significant improvements were seen for even smaller values of ϵ. The algorithm's average error seems to be significantly smaller than that predicted by the user-supplied bound of ϵ. Even for ϵ as high as 3 (implying that a relative error of 300% is tolerated) average relative errors were typically at most 10%, and the true nearest neighbor was found almost half of the time. For many distributions there was relatively little difference in running time and effective performance between different splitting rules, even for the kd-tree, for which upper bounds on search time cannot be proved. However, the BBD-tree retains the efficiency of the kd-tree in these cases, and is more robust for highly clustered data sets, where the kd-tree's performance can be much worse. Finally, the dependencies on dimension and ϵ, seem to be much lower than the bounds presented in Theorem 1.

7 Conclusions.

We have showed that through the use of the BBD-tree, $(1+\epsilon)$-approximate nearest neighbor queries for a set of n points in R^d can be answered in $O(c_{d,\epsilon} \log n)$ time, where $c_{d,\epsilon} \leq d\lceil 1 + 6d/\epsilon \rceil^d$ is a constant depending only on dimension and ϵ. The data structure uses optimal $O(dn)$ space and can be built in $O(dn \log n)$ time. The algorithms we have presented are simple (especially the midpoint splitting rule) and easy to implement. Empirical studies indicate good performance on a number of different point distributions. Unlike many recent results on approximate nearest neighbor searching, the preprocessing is independent of ϵ, and so different levels of precision can be provided from one data structure. Although constant factors in query time grow exponentially with dimension, constant factors in space and preprocessing time grow only linearly in d. We have also shown that the algorithms can be generalized to enumerate approximate k-nearest neighbors in additional $O(kd \log n)$ time. Using auxiliary data structures, it is possible to handle point insertions and deletions in $O(\log n)$ time each.

A somewhat simplified version of the BBD-tree has been implemented in C++. The software is available on the web from `http://www.cs.umd.edu/~mount/ANN/`.

There are a number of important open problems that remain. One is that of improving constant factors for query time. Given the practical appeal of a data structure of optimal $O(dn)$ size for large data sets, an important question is what lower bounds can be established for approximate nearest neighbor searching using data structures of this size. Another question is whether the approximate kth nearest neighbor can be computed in time that is polylogarithmic in both n and k.

8 Acknowledgements.

We would like to thank Michiel Smid for his helpful comments. We would also like to thank the reviewers for a number of very useful comments and suggestions.

References

[AM93a] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22(4):794–806, 1993.

[AM93b] 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.

[AM93c] 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.

[AM95] S. Arya and D. M. Mount. Approximate range searching. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 172–181, 1995.

[AMN+94] 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.

[AMN95] S. Arya, D. M. Mount, and O. Narayan. Accounting for boundary effects in nearest neighbor searching. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 336–344, 1995.

[BBKK97] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. Annu. ACM Sympos. Principles Database Syst.*, pages 78–86, 1997.

[Ber93] M. Bern. Approximate closest-point queries in high dimensions. *Inform. Process. Lett.*, 45:95–99, 1993.

[Bes95] S. N. Bespamyatnikh. An optimal algorithm for closest pair maintenance. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 152–161, 1995.

[BET93] M. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes in Computer Science*, pages 188–199. Springer-Verlag, 1993.

[BG85] C.-D. Bei and R. M. Gray. An improvement of the minimum distortion encoding algorithm for vector quantization. *IEEE Transactions on Communications*, 33:1132–1133, 1985.

[BKK96] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. 22nd VLDB Conference*, pages 28–39, 1996.

[BWY80] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected-time algorithms for closest point problems. *ACM Transactions on Mathematical Software*, 6(4):563–580, 1980.

[CH67] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Trans. Inform. Theory*, 13:57–67, 1967.

[Cha82] B. Chazelle. A theorem on polygon cutting with applications. In *Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 339–349, 1982.

[Cha97] T. Chan. Approximate nearest neighbor queries revisited. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 352–358, 1997.

[CK92] 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.

[CK95] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, pages 263–272, 1995.

[Cla83] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Proc. 24th Ann. IEEE Sympos. on the Found. Comput. Sci.*, pages 226–232, 1983.

[Cla88] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM Journal on Computing*, 17(4):830–847, 1988.

[Cla94] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 160–164, 1994.

[Cle79] J. G. Cleary. Analysis of an algorithm for finding nearest neighbors in Euclidean space. *ACM Transactions on Mathematical Software*, 5(2):183–192, 1979.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[CS93] S. Cost and S. Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–78, 1993.

[dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 1997.

[DDF+90] S. Deerwester, S. T. Dumals, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latend semantic analysis. *J. Amer. Soc. Inform. Sci.*, 41:391–407, 1990.

[DH73] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, NY, 1973.

[DW82] L. Devroye and T. J. Wagner. Nearest neighbor methods in discrimination. In P. R. Krishnaiah and L. N. Kanal, editors, *Handbook of Statistics*, volume 2. North-Holland, 1982.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[FBF77] 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.

[FBS75] J. H. Friedman, F. Baskett, and L. J. Shustek. An algorithm for finding nearest neighbors. *IEEE Trans. Comput.*, C-24(10):1000–1006, 1975.

[FG88] T. Feder and D. H. Greene. Optimal algorithms for clustering. In *Proc. 20th Annu. ACM Sympos. Theory Comput.*, pages 434–444, 1988.

[FM85] 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.

[FPSSU96] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI Press/Mit Press, 1996.

[Fre85] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14:781–798, 1985.

[Fre93] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 175–194, 1993.

[FSN+95] 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.

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:209–221, 1987.

[GG91] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic, Boston, MA, 1991.

[GK92] L. Guan and M. Kamel. Equal-average hyperplane partitioning method for vector quantization of image data. *Pattern Recognition Letters*, 13:693–699, 1992.

[GR93] I. Galperin and R. L. Rivest. Scapegoat trees. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 165–174, 1993.

[IM98] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, 1998. (to appear).

[Kle97] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimension. In *Proc. 29th Annu. ACM Sympos. Theory Comput.*, pages 599–608, 1997.

[KOR98] 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. (to appear).

[LC94] C.-H. Lee and L.-H. Chen. Fast closest codeword search algorithm for vector quantisation. *IEE Proc.-Vis. Image Signal Process.*, 141:143–148, 1994.

[LJF94] K. I. Lin, H. V. Jagdish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, 1994.

[Mei93] S. Meiser. Point location in arrangements of hyperplanes. *Information and Computation*, 106(2):286–303, 1993.

[MNSW95] D. M. Mount, N. Netanyahu, R. Silverman, and A. Y. Wu. Chromatic nearest neighbor searching: A query sensitive approach. In *Proc. 7th Canad. Conf. Comput. Geom.*, pages 261–266, 1995.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[Riv74] R. L. Rivest. On the optimality of Elias's algorithm for performing best-match searches. In *Information Processing*, pages 678–681. North Holland Publishing Company, 1974.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 71–79, 1995.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison Wesley, Reading, MA, 1990.

[Spr91] R. L. Sproull. Refinements to nearest-neighbor searching. *Algorithmica*, 6:579–589, 1991.

[SSS94] C. Schwarz, M. Smid, and J. Snoeyink. An optimal algorithm for the on-line closest-pair problem. *Algorithmica*, 12:18–29, 1994.

[ST83] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26:362–391, 1983.

[Vai89] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete Comput. Geom.*, 4:101–115, 1989.

[WJ96] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. 12th IEEE Internat. Conf. Data Engineering*, pages 516–523, 1996.

[YY85] A. C. Yao and F. F. Yao. A general approach to d-dimensional geometric queries. In *Proc. 17th Ann. ACM Sympos. Theory Comput.*, pages 163–168, 1985.