It's okay to be skinny, if your friends are fat*

Songrit Maneewongvatana and David M. Mount
Department of Computer Science
University of Maryland
College Park, Maryland 20742
{songrit,mount}@cs.umd.edu

December 18, 1999.

Abstract

The kd-tree is a popular and simple data structure for range searching and nearest neighbor searching. Such a tree subdivides space into rectangular cells through the recursive application of some splitting rule. The choice of splitting rule affects the shape of cells and the structure of the resulting tree. It has been shown that an important element in achieving efficient query times for approximate queries is that each cell should be fat, meaning that the ratio of its longest side to shortest side (its aspect ratio) should be bounded. Subdivisions with fat cells satisfy a property called the packing constraint, which bounds the number of disjoint cells of a given size that can overlap a ball of a given radius. We consider a splitting rule called the sliding-midpoint rule. It has been shown to provide efficient search times for approximate nearest neighbor and range searching, both in practice and in terms of expected case query time. However it has not been possible to prove results about this tree because it can produce cells of unbounded aspect ratio. We show that in spite of this, the sliding-midpoint rule generates subdivisions that satisfy the packing constraint, thus explaining their good performance.

1 Introduction

The design of efficient data structures for geometric queries is a fundamental problem. Of particular interest here are problems of the following form: Given a set S of n *data points* in real d-dimensional space, $\mathbf{R}^d$, preprocess these points into a data structure, so that queries about the data set can be answered efficiently. Two important classes of queries are *nearest neighbor queries* and *range queries*. In the first case a query point $q \in \mathbf{R}^d$ is given, and we wish to report the data point nearest to q. In the second case, we are given a query range Q and asked to report or count the data points that lie within Q. Throughout we will assume that d is a fixed constant, independent of n.

One of the simplest and most popular data structures for answering these sorts of queries is Bentley's kd-tree [Ben75]. A kd-tree is a binary tree that represents a hierarchical subdivision of space using splitting planes that are orthogonal to the coordinate axes. Each node of the kd-tree is associated with a closed rectangular region of space, called a *cell*. Each cell is the product of d

*A preliminary version of this paper appeared at the 4th Annual CGC Workshop on Computational Geometry, Johns Hopkins University, October 1999. The support of the National Science Foundation under grant CCR–9712379 is gratefully acknowledged.

closed intervals along each of the coordinate axes. Define the *size* of a cell to be the length of the longest such interval. The root's cell is associated with a bounding hyperrectangle that contains all the points of S. Each internal node is associated with an axis-orthogonal splitting hyperplane. The points of the cell are partitioned to one side or the other of this hyperplane (and points lying on the hyperplane can be placed on either side). The resulting subcells are the children of the original cell. This splitting process is repeated recursively until the number of points is at most one (or more generally a small constant value).

The exact structure of the tree and the associated spatial subdivision depends on a procedure choice of the *splitting rule*, a procedure that selects the splitting hyperplane at each step. Friedman, Bentley, and Finkel proposed the *standard splitting rule* [FBF77], which is among the mostly widely used rules and will be defined in the next section. They showed kd-trees built with this rule can answer nearest neighbor queries in $O(\log n)$ expected-case time, in any fixed dimension d assuming uniformly distributed data points. Lee and Wong showed that rectangular range search queries can be answered in worst-case $O(dn^{1-1/d})$ time [LW77].

A number of other splitting rules have been proposed for kd-trees, including those by Arya and Fu [AF00], Silva-Filho [Fil81], Sproull [Spr91], and White and Jain [WJ96]. More generally, there are many data structures for answering geometric queries based on hierarchical spatial subdivisions (see Samet [Sam90]). These include the R-tree and its variants [Gut84, SRF87, BKSS90, RKV95] the vp-tree [Yia93], the TV-tree [LJF94], the X-tree [BKK96], the M-tree [CPZ97], the SR-tree [KS97], the BAR-tree [DGK99].

One of the questions raised by the large number of different data structures is that of determining the essential properties that a hierarchical space partition tree should possess in order to provide efficient search times for these query problems. In [AMN+98] the following list of sufficient properties was presented for any binary space partition tree to guarantee that approximate nearest neighbor queries can be answered efficiently. Define the *aspect ratio* of a rectangular cell to be the ratio of the lengths of its longest and shortest sides.

- The tree has $O(n)$ nodes, $O(\log n)$ depth and each node of the tree is associated with at least one data point.
- The cells have bounded aspect ratio of cells.
- Cell complexity is a function only of dimension. (This requirement is so that the distance from the query point to a cell can be computed efficiently.)

In particular, they showed that in any fixed dimension d given a tree satisfying the above requirements[1] $(1+\epsilon)$-approximate nearest neighbor queries can be answered in $O((1/\epsilon^d)\log n)$ time. In [AM95] it was shown that from these same properties, $(1+\epsilon)$-approximate range counting queries could be answered in $O(\log n + (1/\epsilon)^d)$ time. The standard kd-tree does not satisfy these properties (it may produce cells of unbounded aspect ratio). Arya et al. introduced the BBD-tree, which satisfies all of these properties [AMN+98]. However, the BBD-tree has the unpleasant property of producing nonconvex cells. Duncan, Goodrich and Kobourov introduced the BAR-tree, which satisfies these properties and also has convex cells [DGK99]. Recently, Dickerson, Duncan and Goodrich have announced that polylogarithmic query times are possible for approximate queries using a new kd-tree splitting rule [DDG99].

[1]The conditions given in [AMN+98] were somewhat more complicated, because they applied to more complex data structure, the BBD-tree. However, this subset of conditions is sufficient for kd-trees.

However, both the BBD-tree and the BAR-tree have elements that make them significantly more complex than kd-trees. The BBD-tree has cells with disconnected boundaries (each cell is the set theoretic difference of two rectangles, one enclosed within the other), and the BAR-tree uses splitting planes that are not axis-orthogonal. This raises the question of whether similar results can be proved for simpler data structures, and the kd-tree in particular. In this paper, we consider a splitting rule, called the *sliding-midpoint* splitting rule. This rule was originally created for the ANN library as an alternative to the standard kd-tree splitting rule for clustered data sets [MA97]. It was analyzed empirically in [MM99], where it was shown to be significantly more efficient than the standard kd-tree for a number of clustered distributions. Arya and Fu showed that this method is efficient for approximate nearest neighbor searching in the expected case [AF00]. We discuss its basic properties below in Section 2. However, the sliding-midpoint tree can produce cells with unbounded aspect ratio, so called *skinny cells*.

Our main result is that the presence of cells of unbounded aspect ratio in the sliding-midpoint tree need not be an impediment to the efficient processing of queries. In [AMN+98] it is shown that the search efficiency depends on a result, called the *packing constraint*. It states that the number of pairwise disjoint cells of size at least s that can overlap a ball of radius r in dimension d is bounded by $O(r/s)^d$. This is easy to prove for subdivisions in which all the cells are *fat* (have bounded aspect ratio.) But the sliding-midpoint method does not have this property. We prove that even so, the sliding-midpoint subdivision satisfies the packing constraint.

This result provides a somewhat deeper insight into the essential properties of hierarchical spatial subdivisions, and the extent to which these properties can be relaxed. It shows that a certain degree of skinniness can be tolerated without significantly degrading efficiency. The key property that the sliding-midpoint rule possesses is that every cell that is skinny along some dimension has a neighbor that is fat along this same dimension. Thus each skinny cell has a fat friend. As a result, it is not possible to create a large number of parallel skinny cells, like sheets of paper in a book. The absence of many parallel skinny cells is the key to its efficiency.

2 Splitting Rules

In this section we consider three splitting rules. We begin with two well-known rules. Recall that we have a rectangular cell in $\mathbf{R}^d$ and an associated set of data points, which lie within the cell.

Standard split: The splitting dimension is chosen to be the one for which the data points have the maximum *spread* (difference between the maximum and minimum values), and the splitting value is chosen to be the coordinate median of the points in that dimension. Friedman, Bentley and Finkel introduced this splitting rule in their definition of the *optimized kd-tree* [FBF77]. This is perhaps the most well-known and widely used splitting method for kd-trees.

Midpoint split: The splitting hyperplane passes through the center of the cell and bisects the longest side of the cell. If there are many sides of equal length, any may be chosen first, say, the one with the lowest coordinate index. If the root cell is a hypercube, then this is just a binary version of the well-known quadtree and octree decompositions.

Observe that in the standard splitting rule, roughly half of the data points are associated with each child. This implies that the tree has $O(\log n)$ depth and $O(n)$ nodes. However, cells may have very high aspect ratio (as can be seen in Fig. 3 below). On the other hand, the midpoint tree has the feature that for all cells, the ratio of the longest to shortest side (the *aspect ratio*) is at most 2.

Unfortunately, if the data are clustered, it is possible to have many *empty cells*, which contain no data points. This is not uncommon in practice, and for highly clustered point sets may result in trees whose size cannot be bounded as a function of n.

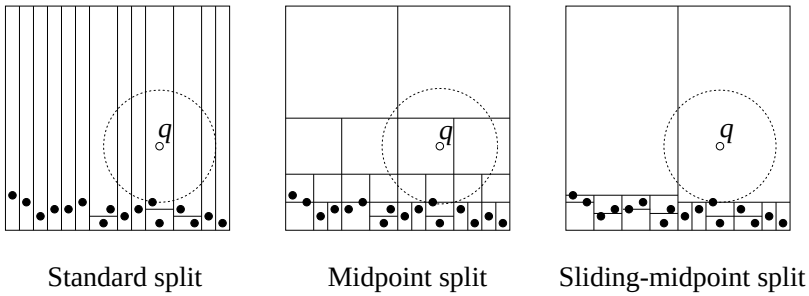

Figure 1: Splitting methods with clustered point sets.

We consider a variant of the midpoint splitting rule, which is designed to provide the same simplicity that makes kd-trees popular, while overcoming some of the shortcomings in the above splitting methods. To understand the problem, suppose that the data points are highly clustered along a few dimensions but vary greatly along some the others (see Fig. 1). The standard kd-tree splitting method will repeatedly split along the dimension in which the data points have the greatest spread, leading to many cells with high aspect ratio. In nearest neighbor processing it is necessary to visit all the leaf cells that overlap the nearest neighbor ball, since any one of them might contain the nearest neighbor. A nearest neighbor query near the center of the bounding square would visit a large number of these cells. The following rule is designed to overcome this.

Sliding-midpoint: First a midpoint split is attempted, by considering a hyperplane passing through the center of the cell and bisecting the cell's longest side. If the data points lie on both sides of the splitting plane then the splitting plane remains here. However, if all the data points lie to one side of the splitting plane, then splitting plane "slides" towards the data points until it encounters the first such point. One child is a leaf cell containing this single point, and the algorithm recurses on the remaining points.

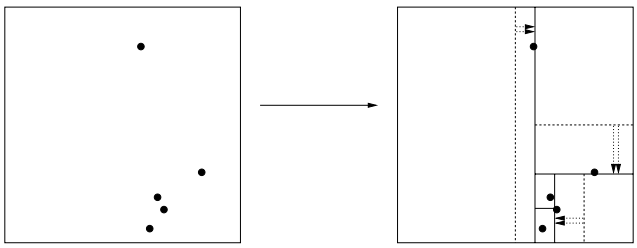

Figure 2: Example of the sliding-midpoint rule.

Fig. 2 illustrates this process on a small example. Fig. 3 shows the result of applying all three rules to a large clustered data set in the plane. The presence of cells of high aspect ratio is quite evident in the standard splitting rule, but much less so in the sliding-midpoint rule. The midpoint split only produces cells of bounded aspect ratio, but there are many empty cells. The problem of empty cells becomes more extreme in higher dimensional spaces.

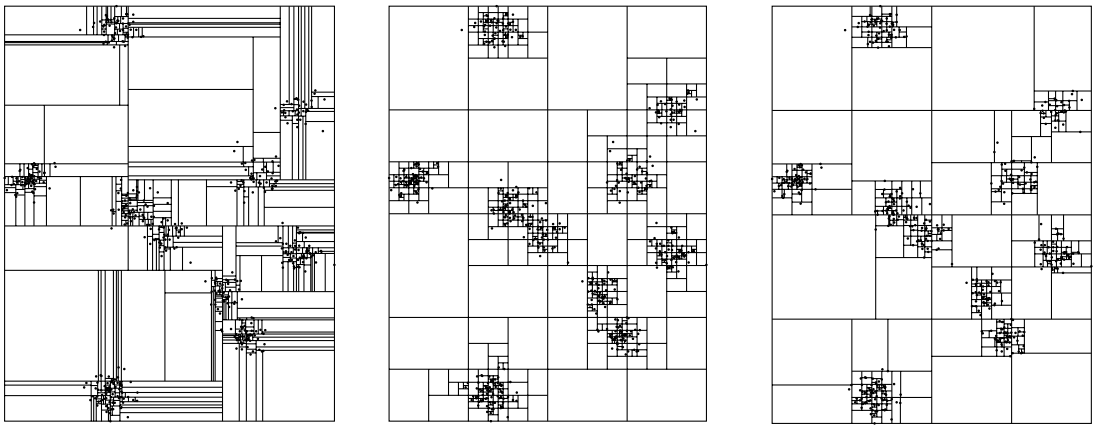

Figure 3: Examples of splitting rules on a common data set.

The sliding-midpoint splitting rule was first introduced in the ANN library for approximate nearest neighbor searching [MA97] and was subsequently analyzed empirically in [MM99]. This method produces no empty nodes, and hence the tree has $O(n)$ nodes. Although cells may not have bounded aspect ratio, observe that for every cell that skinny along some dimension, its sibling cell is fat along this same dimension. Below we show that this is sufficient to satisfy the necessary packing constraint that fat subdivisions possess. As with the standard kd-tree, this tree can be constructed in $O(dn \log n)$ time [MM99].

Because there is no guarantee that the point partition is balanced, the depth of the resulting tree may exceed $O(\log n)$. In theory this deficiency could be remedied by introducing more complex splitting methods [AF00] or through auxiliary data structures that provide rapid access to unbalanced trees [Fre93, ST83]. However this additional complexity does not seem to be warranted in practice. In our experience with real data sets in higher dimensions, we have observed that the depth of the search tree (which is typically $O(\log n)$) seems to be less of a dominating factor in running time than the number leaves visited in the search (which is typically grows exponentially with dimension, and can be heavily influenced by aspect ratio).

3 Packing Lemma for Sliding-Midpoint

In this section we prove that kd-trees built using sliding-midpoint satisfy property the *packing constraint*. For any positive integer m, define a d-dimensional L_m *Minkowski ball* of radius r centered at some point $c \in \mathbf{R}^d$ to be the open set of points $x \in \mathbf{R}^d$ such that

$$\left(\sum_{i=1}^{d} (x_i - c_i)^m \right)^{1/m} < r.$$

For example, an L_2 ball is just the interior of a hypersphere in $\mathbf{R}^d$. In the proof we will use the (easily verified) fact that any L_m ball of radius r can be enclosed within a hypercube of side length $2r$.

Lemma 3.1. (Packing Constraint) *Consider any set C of cells of the sliding-midpoint tree with pairwise disjoint interiors, each of size at least s and each intersecting a Minkowski ball B of radius r. Then C is of size at most $d\left(1+\lceil\frac{4r}{s}\rceil\right)^d$.*

Before giving the proof, we note that except for the leading factor of d, this is essentially the same packing bound proved in [AM95] for the BBD-tree.

Proof

In [AMN+98] it is shown that the number of axis-aligned rectangles with pairwise disjoint interiors and of side length at least $s/2$ that can overlap a Minkowski ball of radius r (or generally any object that can be enclosed within a hypercube of side length $2r$) is at most $(1+\lceil 4r/s\rceil)^d$. We will replace each cell $c \in C$ with a hypercube of side length $s/2$ that intersects B. We will show that these hypercubes can be partitioned into d classes so that no two hypercubes in the same class intersect one another. The result follows by applying the above fact to the hypercubes in each class.

Consider any cell $c \in C$. A side of c is *long* if its length is at least $s/2$ and *short* otherwise. Our approach will be to define a top-down process, which starts at the root cell of the kd-tree. We number the classes from 1 to d. Throughout the process's execution we will maintain the invariant that if a cell's hypercube is assigned to class i, then this cell is long along dimension i.

Consider a cell c that overlaps B, is not in C, but has a descendent in C. Since c has a descendent in C and all the cells of C are of size at least s, it follows that the longest side of c (the side to be split) is of length at least s. Thus, at least one of its children will be long along its splitting side.

First suppose that c is subdivided along some dimension i in such a way that both children are long along dimension i. (This will certainly happen for a midpoint split, but might not happen for a sliding split.) Let c_0 and c_1 be c's children. Since both children are long along dimension i, we may place the hypercubes for the subtree rooted at c_0 so that they lie on one side of the splitting hyperplane and the hypercubes for the subtree rooted at c_1 to lie on the other. Thus, there will be no overlap between these two sets of hypercubes. If c_0 and/or c_1 are in C, then we assign them to class i and place their hypercubes so they overlap B. This satisfies the above invariant. See Fig. 4(a).

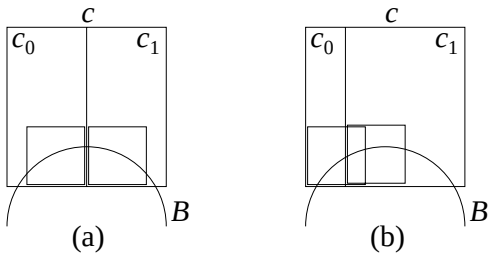

Figure 4: Proof of the packing constraint.

On the other hand, suppose that c is subdivided along dimension i such that one child c_0 is short along dimension i (and hence the other c_1 must be long). This can only arise as the result of a sliding operation, implying that c_1 is a leaf cell. If c_1 is in C, then we assign its hypercube to class i and place it so that it lies to one side of the splitting hyperplane, and overlaps the ball. By our invariant no cells of C in the subtree rooted at c_0 will be assigned to class i (since they are all short along this side). Therefore there is no possibility of overlap within the same class between c_1's

hypercube and any hypercubes nested within c_0. If c_0 is in C, then by definition of C its longest side j is of length at least s. Assign it to class j. This satisfies the above invariant.

In summary, by applying this process top-down to the kd-tree, we can replaced each cell in C with a hypercube and an associated class, such that no two hypercubes in the same class intersect one another, and all hypercubes intersect B. Applying the rectangle packing result from [AMN+98] to each class completes the proof. □

4 Concluding Remarks

We have shown that the sliding-midpoint kd-tree, satisfies the packing constraint, in spite of the fact that its cells are not of bounded aspect ratio. The worst-case analyses appearing in [AMN+98] for approximate nearest neighbor searching and [AM95] for approximate range searching, are based on only two properties of the BBD-tree data structure, its $O(\log n)$ depth and the packing constraint. Although the sliding-midpoint tree is not necessarily of logarithmic depth, in our practical experience with the data structure we found that the depth of the tree was $O(\log n)$ for all data sets we tested [MM99]. Furthermore, empirically measured running times tended to depend much more heavily on the number of leaf cells visited than on any other aspect of the tree. Thus, the results of this paper helps to explain the good practical performance of this splitting rule.

We suspect that kd-trees and their variants will continue to be popular data structures in practice because of their simplicity and flexibility. Given this, an obvious open problem is to determine the splitting rules that provide the best performance for both exact and approximate queries. This study provides some insight as to what properties such a tree should possess.

5 Acknowledgements

We would like to thank Sunil Arya for many helpful conversations and suggestions.

References

[AF00] S. Arya and H. Y. Fu. Expected-case complexity of approximate nearest neighbor searching. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, 2000. To appear.

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

[AMN+98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45:891–923, 1998.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[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.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 322–331, 1990.

[CPZ97] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. 23rd VLDB Conference*, pages 426–435, 1997.

[DDG99] M. Dickerson, C. A. Duncan, and M. T. Goodrich. Personal communication, 1999.

[DGK99] C. Duncan, M. Goodrich, and S. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 300–309, 1999.

[FBF77] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Software*, 3(3):209–226, 1977.

[Fil81] Y. V. Silva Filho. Optimal choice of discriminators in a balanced K-D binary search tree. *Information Processing Letters*, 13:67–70, 1981.

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

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 47–57, 1984.

[KS97] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 369–380, 1997.

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

[LW77] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Inform.*, 9:23–29, 1977.

[MA97] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. Center for Geometric Computing 2nd Annual Fall Workshop on Computational Geometry, URL: `http://www.cs.umd.edu/~mount/ANN`, 1997.

[MM99] S. Maneewongvatana and D. Mount. Analysis of approximate nearest neighbor searching with clustered point sets. In *ALENEX*, 1999.

[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. F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multidimensional objects. In *Proc. 13th VLDB Conference*, pages 507–517, 1987.

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

[WJ96] D. A. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Lab., Univ. California, San Diego, CA, 1996.

[Yia93] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 311–321, 1993.