The Skip Quadtree: A Simple Dynamic Data Structure for Multidimensional Data

David Eppstein Michael T. Goodrich [†] Jonathan Z. Sun [†]

Department of Computer Science
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425, USA
{eppstein,goodrich,zhengsun}(at)ics.uci.edu

ABSTRACT

We present a new multi-dimensional data structure, which we call the skip quadtree (for point data in $\mathbf{R}^2$) or the skip octree (for point data in $\mathbf{R}^d$, with constant $d > 2$). Our data structure combines the best features of two well-known data structures, in that it has the well-defined "box"-shaped regions of region quadtrees and the logarithmic-height search and update hierarchical structure of skip lists. Indeed, the bottom level of our structure is exactly a region quadtree (or octree for higher dimensional data). We describe efficient algorithms for inserting and deleting points in a skip quadtree, as well as fast methods for performing point location, approximate range, and approximate nearest neighbor queries.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*geometrical problems and computations*; H.3.3 [**Information Systems**]: Information Storage and Retrieval—*information search and retrieval*

General Terms

Algorithms, Theory

Keywords

Skip quadtree, quadtree, octree, dynamic data structure, point location, range, nearest neighbor, approximation algorithm.

1. INTRODUCTION

Data structures for multidimensional point data are of significant interest in the computational geometry, computer graphics, and scientific data visualization literatures. They allow point data to be stored and searched efficiently, for example to perform range queries to count or report (possibly approximately) the points that are contained in a given query region. We are interested in this paper in data structures for multidimensional point sets that are dynamic, in that they allow for fast point insertion and deletion, as well as efficient, in that they use linear space and allow for fast query times.

[†]Supported by NSF Grants CCR-0225642, CCR-0311720, CCR-0312760, and DUE-0231467.

1.1 Related Previous Work

Linear-space multidimensional data structures typically are defined by hierarchical subdivisions of space, which give rise to tree-based search structures. That is, a hierarchy is defined by associating with each node v in a tree T a region $R(v)$ in $\mathbf{R}^d$ such that the children of v are associated with subregions of $R(v)$ defined by some kind of "cutting" action on $R(v)$. Examples include:

- *quadtrees* [32]: regions are defined by squares in the plane, which are subdivided into four equal-sized squares for any regions containing more than a single point. (These are also called PR quadtrees, and we always refer to this variant of quadtrees in this paper.) So each internal node in the underlying tree has four children and regions have optimal aspect ratios (which is useful for many types of queries). Unfortunately, the tree can have arbitrary depth, independent even of the number of input points. Even so, point insertion and deletion is fairly simple.
- *octrees* [22, 32]: regions are defined by hypercubes in $\mathbf{R}^d$, which are subdivided into 2^d equal-sized hypercubes for any regions containing more than a single point. So each internal node in the underlying tree has 2^d children and, like quadtrees, regions have optimal aspect ratios and point insertion/deletion is simple, but the tree can have arbitrary depth.
- *k-d trees* [10]: regions are defined by hyperrectangles in $\mathbf{R}^d$, which are subdivided into two hyperrectangles using an axis-perpendicular cutting hyperplane through the median point, for any regions containing more than two points. So the underlying tree can be made a balanced binary tree with $\lceil \log n \rceil$ depth for a static set of points. Unfortunately, the regions can have arbitrarily large aspect ratios, which can adversely affect the efficiencies of some queries. Although some variants were proposed to help, such as by cutting along the longest side instead of along each axis recursively [15], they only improve the aspect ratios experimentally or statistically. In addition, maintaining an efficient k-d tree subject to point insertions and removal is non-trivial.
- *compressed quad/octrees* [2, 11–13]: regions are defined in the same way as in a quadtree or octree (depending on the dimensionality), but paths in the tree consisting of nodes with only one non-empty child are compressed to single edges.

This compression allows regions to still be hypercubes (with optimal aspect ratio), but it changes the subdivision process from a four-way cut to a reduction to at most four disjoint hypercubes inside the region. It also forces the height of the (compressed) quad/octree to be at most $O(n)$. This height bound is still not very efficient, of course.

- *balanced box decomposition (BBD) trees* [6–8]: regions are defined by hypercubes with smaller hypercubes subtracted away, so that the height of the decomposition tree is $O(\log n)$. These regions have good aspect ratios, that is, they are "fat" [20,21], but they are not convex, which limits some of the applications of this structure. In addition, making this structure dynamic appears non-trivial.

- *balanced aspect-ratio (BAR) trees* [17, 19]: regions are defined by convex polytopes of bounded aspect ratio, which are subdivided by hyperplanes perpendicular to one of a set of $2d$ "spread-out" vectors so that the height of the decomposition tree is $O(\log n)$. This structure has the advantage of having convex regions and logarithmic depth, but the regions are no longer hyperrectangles (or even hyperrectangles with hyperrectangular "holes"). In addition, making this structure dynamic appears non-trivial.

This summary is, of course, not a complete review of existing work on space partitioning data structures for multidimensional point sets. The reader interested in further study of these topics is encouraged to read the books by de Berg *et al.* [14] and Samet [34,35,38], as well as the book chapters by Asano *et al.* [9], Samet [36,37,39], Lee [24], Aluru [1], Naylor [29], Nievergelt and Widmayer [31], Leutenegger and Lopez [25], Duncan and Goodrich [18], and Arya and Mount [5].

1.2 Our Results

In this paper we present a dynamic data structure for multidimensional data, which we call the *skip quadtree* (for point data in $\mathbf{R}^2$) or the *skip octree* (for point data in $\mathbf{R}^d$, for fixed $d > 2$). For the sake of simplicity, however, we will often use the term "quadtree" to refer to both the two- and multi-dimensional structures. This structure provides a hierarchical view of a quadtree in a fashion reminiscent of the way the skip-list data structure [27,33] provides a hierarchical view of a linked list. Our approach differs fundamentally from previous techniques for applying skip-list hierarchies to multidimensional point data [26,30] or interval data [23], however, in that the bottom-level structure in our hierarchy is not a list—it is a tree. Indeed, the bottom-level structure in our hierarchy is just a compressed quadtree [1, 2, 11–13]. Thus, any operation that can be performed with a quadtree can be performed with a skip quadtree. More interestingly, however, we show that point location, approximate range, and approximate nearest neighbor queries can be performed in a skip quadtree in $O(\log n)$, $O(\varepsilon^{1-d} + \log n)$, and $O(\varepsilon^{1-d}(\log n + \log \varepsilon^{-1}))$ time, respectively, for any constant $\varepsilon > 0$. We also show that point insertion and deletion can be performed in $O(\log n)$ time. We describe both randomized and deterministic versions of our data structure, with the above time bounds being expected bounds as well as bounds with high probability (w.h.p.) for the randomized version and worst-case bounds for the deterministic version.

Due to the balanced aspect ratio of their cells, quadtrees have many geometric applications including range searching, proximity problems, construction of well separated pair decompositions, and quality triangulation. However, due to their potentially high depth, maintaining quadtrees directly can be expensive. Our skip quadtree data structure provides the benefits of quadtrees together with fast update and query times even in the presence of deep tree branches, and is, to our knowledge, the first balanced aspect ratio subdivision with such efficient update and query times. We believe that this data structure will be useful for many of the same applications as quadtrees. In this paper we demonstrate the skip quadtree's benefits for three simple types of queries: point location within the quadtree itself, approximate range counting, and approximate nearest neighbor searching.

2. PRELIMINARIES

In this section we discuss some preliminary conventions we use in this paper.

2.1 Notational Conventions

Throughout this paper we use Q for a quadtree and p, q, or r for squares or quadrants of squares associated with the nodes of Q. We use S with $|S| = n$ for a set of data points in $\mathbf{R}^d$ upon which the quadtree is built, and x, y, z for the points in S. We use u or v to denote a point location in $\mathbf{R}^d$ and $p(u)$ for the smallest square in Q that covers the location, regardless if u is in the underlying point set S for Q or not. Constant d is reserved for the dimensionality of our search space, $\mathbf{R}^d$, and we assume throughout that $d \geq 2$ is a constant. In d-dimensional space we still use the term "square" to refer to a d-dimensional cube and we use "quadrant" for any of the $1/2^d$ partitions of a square r into squares having the center of r as a corner and sharing part of r's boundary. A square r is identified by its center $c(r)$ and its half side length $s(r)$.

2.2 The Computational Model

As is standard practice in computational geometry algorithms dealing with quadtrees and octrees (e.g., see [12]), we assume in this paper that certain operations on points in $\mathbf{R}^d$ can be done in constant time. In real applications, these operations are typically performed using hardware operations that have running times similar to operations used to compute linear intersections and perform point/line comparisons. Specifically, in arithmetic terms, the computations needed to perform point location in a quadtree, as well as update, range query, or nearest neighbor query operations, involve finding the most significant binary digit at which two coordinates of two points differ. This can be done in $O(1)$ machine instructions if we have a most-significant-bit instruction, or by using floating-point or extended-precision normalization. If the coordinates are not in binary fixed or floating point, such operations may also involve computing integer floor and ceiling functions.

2.3 The Compressed Quadtree

As the bottom-level structure in a skip quadtree is a compressed quadtree [1, 2, 11–13], let us briefly review this structure.

The compressed quadtree is defined in terms of an underlying (standard) quadtree for the same point set; hence, we define the compressed quadtree by identifying which squares from the standard quadtree should also be included in the compressed quadtree. Without loss of generality, we can assume that the center of the root square (containing the entire point set of interest) is the origin and the half side length for any square in the quadtree is a power of 2. A point x is contained in a square p iff $-s(p) \leq x_i - c(p)_i < s(p)$ for each dimension $i \in [1, \cdots, d]$. According to whether $x_i - c(p)_i < 0$ or ≥ 0 for all dimensions we also know in which quadrant of p that x is contained.

Define an *interesting square* of a (standard) quadtree to be one that is either the root of the quadtree or that has two or more nonempty quadrants. Then it is clear that any quadtree square p containing two or more points contains a unique largest interesting square q (which is either p itself or a descendent square of p in the standard quadtree). In particular, if q is the largest interesting square for p, then q is the lowest common ancestor (LCA) in the quadtree of the points contained in p. We compress the (standard) quadtree to explicitly store only the interesting squares, by splicing out the non-interesting squares and deleting their empty children from the original quadtree. That is, for each interesting square p, we store 2^d bi-directed pointers one for each d-dimensional quadrant of p. If the quadrant contains two or more points, the pointer goes to the largest interesting square inside that quadrant; if the quadrant contains one point, the pointer goes to that point; and if the quadrant is empty, the pointer is NULL. We call this structure a *compressed quadtree* [1,2,11–13]. (See Fig. 1.)

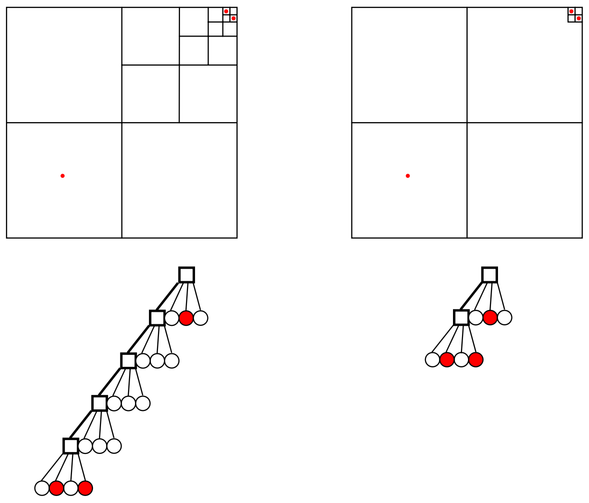

Figure 1: A quadtree containing 3 points (left) and its compressed quadtree (right). Below them are the pointer representations, where a square or an interesting square is represented by a square, a point by a solid circle and an empty quadrant by a hollow circle. The 4 children of each square are ordered from left to right according to the I, II, III, IV quadrants.

A compressed d-dimensional quadtree Q for n points has size $O(n)$, but its worst-case height is $O(n)$, which is inefficient yet nevertheless improves the arbitrarily-bad worst-case height of a standard quadtree. These bounds follow immediately from the fact that there are $O(n)$ interesting squares, each of which has size $O(2^d)$.

With respect to the arithmetic operations needed when dealing with compressed quadtrees, we assume that we can do the following operations in $O(1)$ time:

- Given a point location u and a square p, decide if p covers u and if yes, which quadrant of p covers u.
- Given a quadrant of a square p containing two points x and y, find the largest interesting square inside this quadrant.
- Given a quadrant of p containing an interesting square r and a point $x \notin r$, find the largest interesting square inside this quadrant intersecting both r and x.

A standard point location search in a compressed quadtree Q is to locate the smallest quadtree square covering a query location u. Such a search starts from the quadtree root and follows the parent-child pointers, and returns the required interesting square $p(u)$. Note that $p(u)$ is either a leaf node of Q or an internal node with none of its child nodes covering the location of u. If the quadrant of $p(u)$ covering the location of u stores exactly one point that matches u, then we find u in the underlying data set S. If a point other than u or a smaller interesting square instead of a point is stored in that quadrant, then u is not in S. The search proceeds in a top-down fashion from the root, taking $O(1)$ time per step; hence, the search time is $O(n)$.

Inserting a new point x starts by locating the interesting square $p(x)$ covering the location of x. Inserting x into an empty quadrant of $p(x)$ only takes $O(1)$ pointer changes. If the quadrant of $p(x)$ that x is inserted into already contains a point y or an interesting square r, then we insert to Q a new interesting square $q \subset p$ that contains both x and y (or r) but separates x and y (or r) into different quadrants of q. This can be done in $O(1)$ time. So the insertion time is $O(1)$, given $p(x)$.

Deleting x may cause its covering interesting square $p(x)$ to be no longer interesting. If this happens, we splice $p(x)$ out and delete its empty children from Q. Note that the parent node of $p(x)$ is still interesting, since deleting x doesn't change the number of nonempty quadrants of the parent of $p(x)$. Therefore, by splicing out at most one node (with $O(1)$ pointer changes), the compressed quadtree is updated correctly. So a deletion also takes $O(1)$ time, given $p(x)$. The following theorem is implied in related previous works, e.g., [1,2,11–13].

THEOREM 1. *Point-location search, as well as point insertion and deletion, in a compressed d-dimensional quadtree of n points can be done in $O(n)$ time.*

Thus, the worst-case time for querying a compressed quadtree is no better than that of brute-force searching of an unordered set of points. Still, like a standard quadtree, a compressed quadtree is unique given a set of n points and a (root) bounding box, and this uniqueness allows for constant-time update operations if we have already identified the interesting square involved in the update. Therefore, if we could find a faster way to query a compressed quadtree while still allowing for fast updates, we could construct an efficient dynamic multidimensional data structure.

3. THE RANDOMIZED SKIP QUADTREE

In this section, we describe and analyze the randomized skip quadtree data structure, which provides a hierarchical view of a compressed quadtree so as to allow for logarithmic time (in expectation and w.h.p.) querying and updating, while keeping the expected space bound linear.

3.1 Randomized Skip Quadtree Definition

The randomized skip quadtree is defined by a sequence of compressed quadtrees that are respectively defined on a sequence of subsets of the input set S. In particular, we maintain a sequence of subsets of the input points S, such that $S_0 = S$, and, for $i > 0$, S_i is sampled from S_{i-1} by keeping each point with probability 1/2. (So, with high probability, $S_{2\log n} = \emptyset$.) For each S_i, we form a (unique) compressed quadtree Q_i for the points in S_i. We therefore view the Q_i's as forming a sequence of levels in the skip quadtree, such that S_0 is the bottom level (with its compressed quadtree defined for the entire set S) and S_{top} being the top level, defined as the lowest level with an empty underlying set of points.

Note that if a square p is an interesting square in Q_i, then it is also an interesting square in the lower level Q_{i-1}. Indeed, this *coherence* property between levels in the skip quadtree is what facilitates fast searching. For each interesting square p in a compressed quadtree Q_i, we add two pointers in addition to the parent-child pointers described in Section 2.3: one to the same square p in Q_{i-1} and another to the same square p in Q_{i+1} if p exists in Q_{i+1}, or NULL otherwise. The sequence of Q_i's and S_i's, together with these auxiliary pointers define the skip quadtree. (See Fig. 2.)

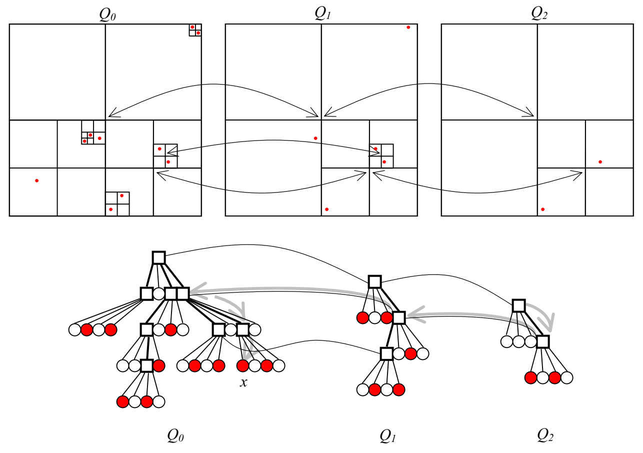

Figure 2: A randomized skip quadtree consists of Q_0, Q_1 and Q_2. (Identical interesting squares in two adjacent compressed quadtrees are linked by a double-head arrow between the square centers.)

3.2 Search, Insertion, and Deletion in a Randomized Skip Quadtree

To find the smallest square in Q_0 covering the location of a query point u, we start with the root square $p_{l,start}$ in Q_l. (l is the largest value for which S_l is nonempty and l is $O(\log n)$ w.h.p.) Then we search u in Q_l as described in Section 2.3, following the parent-child pointers until we stop at the smallest interesting square $p_{l,end}$ in Q_l that covers the location of u. After we stop searching in each Q_i we go to the copy of $p_{i,end}$ in Q_{i-1} and let $p_{i-1,start} = p_{i,end}$ to continue searching in Q_{i-1}. (See the searching path of x in Fig. 2.)

LEMMA 2. *For any point location u, the expected number of searching steps within any individual level Q_i is constant. The total number of searching steps through all levels is $O(\log n)$ in expectation and w.h.p.*

PROOF. Our analysis directs to the worst combined choice of Q_i and u. Suppose the searching path of u in Q_i from the root of Q_i is $p_0, p_1, \cdots, p_m$. Consider the probability $Pr(j)$ of $Event(j)$ such that p_{m-j} is the last one in $p_0, p_1, \cdots, p_m$ which is also interesting in Q_{i+1}. (Note that $Event(j)$ and $Event(j')$ are excluding for any $j \neq j'$.) Then j is the number of searching steps that will be performed in Q_i.

Consider an example that each p_{m-j}, $j = 0, \cdots, m$, has exactly two non-empty quadrants, and the non-empty quadrant of p_{m-j} hanging off the path $p_0, p_1, \cdots, p_m$ contains exactly one point x_{m-j}. And the lowest square p_m is a leaf of Q_i containing two points x_m and x_{m+1}. (See Fig. 3.) $Event(j)$ happens iff x_{m-j} is selected to S_{i+1}, and exactly one point among the $j+1$ points $x_{m-j+1}, \cdots, x_{m+1}$ is also selected. The expected value of j is then

$$\begin{aligned} E(j) &= \sum_{j=1}^{m} jPr(j) = \sum_{j=1}^{m} j \cdot \tfrac{1}{2} \cdot \tfrac{j+1}{2^{j+1}} = \tfrac{1}{4}\left(\sum_{j=1}^{m} \tfrac{j^2}{2^j} + \sum_{j=1}^{m} \tfrac{j}{2^j}\right) \\ &\approx \tfrac{1}{4}(6.0 + 2.0) = 2.0. \end{aligned} \tag{1}$$

(See Appendix A for the computation of the progressions.) We also argue that the above analysis gives the biggest possible $E(j)$. (See Appendix B.) Therefore for the worst choice of Q_i and u, the expectation of j is no more than 2. This adds up to the expectation of $O(\log n)$ for the total number of searching steps through all levels, since the expected number of non-empty subsets is $O(\log n)$ as each S_i contains half of the points in the lower level S_{i-1}. For a high probability analysis, we still consider the above example and backtrack the searching path of u through all levels. In each Q_i we flip a coin for each point $x_{m+1}, x_m, \cdots$ to decide if it is selected into S_{i+1}. The selection of any two points leads to the promotion of a square to Q_{i+1} (i.e., at this square the searching path stops in Q_{i+1} and jumps to Q_i). So it becomes the problem of how many coin flips is needed to get $2c \log n$ heads (promotions) for a constant c. We can get $O(\log n)$ w.h.p. by applying a Chernoff bound. (See Appendix C.) □

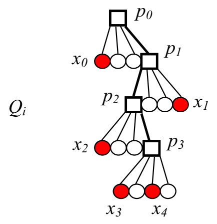

Figure 3: The example structure of Q_i for computing $E(j)$. Say $j = 2$ for example. Sqare $p_1 = p_{m-j}$ is the lowest square on this path which is still interesting in Q_{i+1} iff x_1 and exactly one point among $\{x_2, x_3, x_4\}$ are selected to S_{i+1}.

To insert a point x into the structure, we perform the above point location search which finds $p_{i,end}$ within all the Q_i's, flip coins to find out which S_i's x belongs to, then for each S_i containing x, insert x into $p_{i,end}$ in Q_i as described in Section 2.3. Note that by flipping coins we may create one or more new non-empty subsets $S_{l+1}, \cdots$ which contains only the point x, and we shall consequently create the new compressed quadtrees $Q_{l+1}, \cdots$ containing only x and add them into our skip data structure. Deleting a point x is similar. We do a search to find $p_{i,end}$ in all Q_i's. Then for each Q_i that contains x, delete x from $p_{i,end}$ in Q_i as described in Section 2.3, and remove Q_i from our data structure if Q_i becomes empty.

THEOREM 3. *Searching a point location and inserting or deleting any point in a randomized d-dimensional skip quadtree of n points takes $O(\log n)$ time in expectation and w.h.p.*

PROOF. The searching time is shown in Lemma 2. For any point to be inserted or deleted, let X be the number of non-empty subsets other than S_0 that contains this point. $E(X)$ is then the expected number of heads in a sequence of coin flips before seeing the first tail, which is $\sum_{i=1}^{\infty} \frac{i}{2^{i+1}} = 1$. (See Appendix A.) X can also be bounded to $O(\log n)$ w.h.p. by applying a Chernoff bound. (See Appendix C.) Therefore the time of searching dominates that of insertion and deletion. □

In addition, note that the expected space usage for a randomized skip quadtree is $O(n)$, since the expected size of the compressed quadtrees in the levels of the skip quadtree forms a geometrically decreasing sum that is $O(n)$.

4. THE DETERMINISTIC SKIP QUADTREE

In the deterministic version of the skip quadtree data structure, we again maintain a sequence of subsets S_i of the input points S with $S_0 = S$ and build a compressed quadtree Q_i for each S_i. However, in the deterministic case, we make each Q_i an ordered tree and sample S_i from S_{i-1} in a different way. We can order the 2^d quadrants of each d-dimensional square (e.g., by the I, II, III, IV quadrants in $\mathbf{R}^2$ as in Fig. 1 or by the lexical order of the d-dimensional coordinates in d dimensional space), and accordingly call the compressed quadtree obeying such order an *ordered compressed quadtree*. We build an ordered compressed quadtree Q_0 for

$S_0 = S$ and let $L_0 = L$ be the ordered list of S_0 in Q_0 from left to right. Next we make a skip list $\mathcal{L}$ for L with L_i being the i-th level of the skip list. Let S_i be the subset of S that corresponds to the i-th level L_i of $\mathcal{L}$. Then we build an ordered compressed quadtree Q_i for each S_i. Let x_i be the copy of x at level i in $\mathcal{L}$ and $p_i(x)$ be the smallest interesting square in Q_i that contains x. Then, in addition to the pointers in Section 3, we put a bi-directed pointer between x_i and $p_i(x)$ for each $x \in S$. (See Fig. 4.)

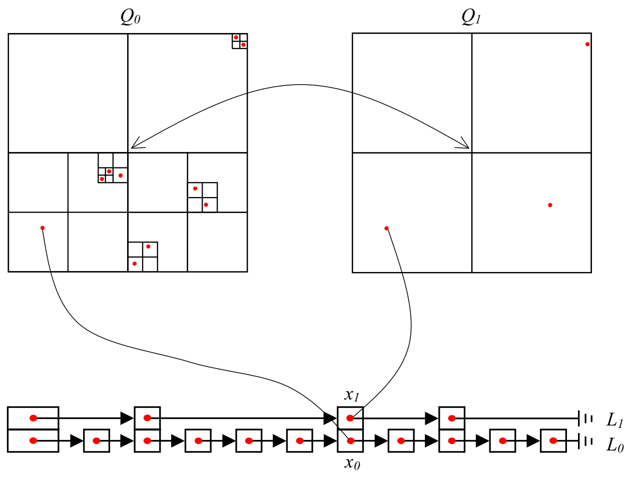

Figure 4: A deterministic skip quadtree guided by a deterministic 1-2-3 skip list.

LEMMA 4. *The order of S_i in Q_i is L_i.*

PROOF. Noting that an interesting square in Q_i is also an interesting square in Q_{i-1}, the lowest common ancestor of two points x and y in Q_i is also a common ancestor of them in Q_{i-1}. Therefore the order of S_i in Q_i is a subsequence of the order of S_{i-1} in Q_{i-1}. Given that the order of S_0 in Q_0 is L_0, the order of S_i in Q_i is L_i by induction. $\square$

The skip list $\mathcal{L}$ is implemented as a deterministic 1-2-3 skip list in [28], which maintains the property that between any two adjacent columns of height i there are 1, 2 or 3 columns of height $i-1$. (See Fig. 4.) Searching in a 1-2-3 skip list takes (worst case) $O(\log n)$ time since there are $O(\log n)$ levels. Insertion and deletion can be done by a search plus $O(1)$ promotions or demotions at each level along the searching path [28].

Lemma 4 provides that the order of points in each Q_i is consistent with L_0. However, in order to guide the skip quadtree by the skip list $\mathcal{L}$, we need not the list L_0 but a total order because search in a skip list requests doing comparisons of keys. We binarize Q_0 by adding $d-1$ levels of dummy nodes, one level per dimension, between each interesting square and its 2^d children. Then we independently maintain a total order (the in-order of the binary Q_0) for the set of interesting squares, dummy nodes and points in Q_0. The order is maintained as in [16] which supports the following operations: 1) insert x before or after some y; 2) delete x; and 3) compare the order of two arbitrary x and y. All operations can be done in worst case $O(1)$ time. These operations give a total order out from a linked list, which suffices the search in $\mathcal{L}$. Because of the binarization of Q_0 and the inclusion of all internal nodes of the binarized Q_0 in our total order, when we insert a point x into Q_0, we get a y (parent of x in the binary tree) before or after the insertion point of x, so that we can accordingly insert x into our total order.

4.1 Search, Insertion, and Deletion in a Deterministic Skip Quadtree

Searching for a point location in a deterministic skip quadtree structure is as in a randomized skip quadtree. However the running time in the deterministic version is as following.

LEMMA 5. *The number of searching steps to locate a point within any individual Q_i is constant in a deterministic skip quadtree.*

PROOF. Suppose the searching sequence (of interesting squares) in Q_i is $p_0, p_1, \cdots, p_m$ with $p_0 = p_{i,start}$ and $p_m = p_{i,end}$. Since each interesting square has at least two non-empty quadrants, there are at least $m+1$ non-empty quadrants hanging off the tree path from p_1 to p_m. The points contained in these quadrants form a consecutive segment in L_i, with furthermore the points contained in each individual quadrant being consecutive. Since $p_1, \cdots, p_m$ are not interesting in Q_{i+1}, at most one among these $\geq m+1$ quadrants is still non-empty in Q_{i+1}, otherwise the LCA of the two non-empty quadrants in Q_i will be interesting in Q_{i+1}. Therefore based on the 1-2-3 property of $\mathcal{L}$, there are at most 7 such non-empty quadrants hanging off the path $p_1, \cdots, p_m$ so that $m \leq 6$. $\square$

To insert or delete a point y into or from S, we first search the quadtree structure to locate y in each Q_i. Then we insert or delete y in the binary Q_0 and update our total order. Then we insert or delete y in the skip list $\mathcal{L}$, referring to the total order. After promoting or demoting any point x from L_i to L_{i+1} or L_{i-1} during the skip list insertion or deletion, we do accordingly an insertion of x in Q_{i+1} or a deletion of x in Q_i. (See Fig. 5.)

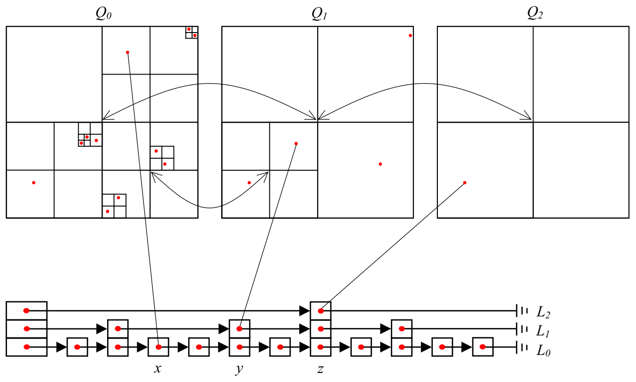

Figure 5: The insertion of x into the deterministic skip quadtree in Fig 4. Inserting x causes the promotions of y and z in $\mathcal{L}$, and consequently the insertion of y into Q_1 and the creation of a new compressed quadtree Q_2 for z.

To delete x from Q_i we go from x_i to the smallest interesting square $p_i(x)$ containing x in Q_i following the pointers. Then the deletion given $p_i(x)$ is as described in Section 2.3. To insert x into Q_{i+1} we go from x_i to $p_i(x)$ in Q_i, then traverse upwards in Q_i until we find the lowest ancestor q of x which is also interesting in Q_{i+1}. (This is the reversed process of searching x in Q_i with $q = p_{i,start} = p_{i+1,end}$ so it takes at most 6 steps by Lemma 5.) Then we go to the same square q in Q_{i+1} and insert x. The insertion of x in Q_{i+1} given q is as described in Section 2.3. Also, as in Section 3 we may create new Q_i or remove empty Q_i during this procedure.

THEOREM 6. *Search, insertion and deletion in a deterministic d-dimensional skip quadtree of n points take worst case $O(\log n)$ time.*

Likewise, the space complexity of a deterministic skip quadtree is $O(n)$.

5. APPROXIMATE RANGE QUERIES

In this section, we describe how to use a skip quadtree to perform approximate range queries, which are directed at counting or reporting the points in S that belong to a query region. A region R is *k-fat* if for any hyper-ball smaller than R and centered at a point location $u \in R$, at least $1/k$ volume of the ball is covered by R. We'll show that a skip quadtree data structure can answer an approximate range query with a convex or non-convex k-fat region in $O(\log n + \varepsilon^{1-d})$ or $O(\log n + \varepsilon^{-d})$ time, which are optimal and match the best previous results.

For simplicity of expression, we assume that the query region is a hyper-sphere (Euclidian ball) throughout this section, however extend the results to any k-fat region at the end of the section. Recall that we use x, y for points in the data set and u, v for arbitrary points (locations) in $\mathbf{R}^d$. A $(1+\varepsilon)$-approximate range query with error $\varepsilon > 0$ is a triple (v, r, ε) that counts all points x with $dist(v,x) \le r$ but also some arbitrary points y with $r < dist(v,y) \le (1+\varepsilon)r$. That is, the query region R is a (hyper-) sphere with center v and radius r, and the permissible error range A is a (hyper-) annulus of thickness εr surrounding R.

5.1 General Idea

Suppose we have a space partition tree T where each tree node is associated with a region in $\mathbf{R}^d$. Given a query (v, r, ε) with region R and annulus A, we call a node $p \in T$ an *in*, *out*, or *stabbing* node if the $\mathbf{R}^d$ region associated with p is contained in $R \cup A$, has no intersection with R, or intersects both R and $\overline{R \cup A}$, respectively. (See Fig. 6.) In order to answer the query, we only need to examine (expand) each stabbing node since for an in node or out node we can simply include or exclude it. That is, the query can be answered by searching all stabbing nodes. A straightforward solution takes $O(|B|D_T)$ time, with B being the set of stabbing nodes and D_T the depth of T. Previously studied space partition trees, such as BBD trees [4,6–8] and BAR trees [17,19], have an upper-bound on $|B|$ of $O(\varepsilon^{1-d})$ and D_T of $O(\log n)$, which is optimal [7]. A more elegant analysis in [7] improved the running time to $O(\log n + \varepsilon^{1-d})$.

We consider the bottom level in the skip structure, Q_0, as our space partition tree T. At the first glance the depth of Q_0 is $O(n)$ and the number of stabbing squares (nodes) can also be $O(n)$ since we can have all nodes in a tree path being nested stabbing squares. However we observe that for a set of nested stabbing squares that cover the same area of $R \cup A$, we only need to examine the smallest one in answering the approximate range query. Given R and A, define a *critical square* $p \in Q_0$ as a stabbing node of Q_i whose child nodes are either not stabbing, or still stabbing but cover less volume of R than p does. (See Fig. 6.) Indeed, searching the set C of critical squares instead of the set B of stabbing squares suffices answering an approximate range query. Next we'll fulfill the query algorithm by bounding the number of critical squares in Q_0 and providing an efficient algorithm for searching these squares via the skip structure.

5.2 Packing Lemmas

The ratio of the unit volume of A, $(\varepsilon r)^d$, to the lower bound of volume of A that is covered by any stabbing node is called the *packing function* ρ of T, a function of n that is often used to bound the size of B. A constant ρ immediately results in the optimal $|B| = O(\varepsilon^{1-d})$ for convex k-fat regions, in which case the total volume of A is $O(\varepsilon r^d)$. BBD trees and BAR trees provide a constant ρ by using delicately designed regions for tree nodes. However, for a compressed quadtree with box-shaped regions, ρ is obviously a constant which depends only on the dimensionality and metric of space. The following packing lemma bounds the size of C to $O(\varepsilon^{1-d})$.

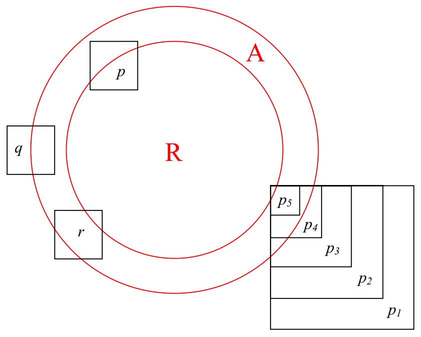

Figure 6: Squares p, q, and r are in, out and stabbing, respectively. Squares $p_1, \cdots, p_4$ (sizes are not drawn to scale) show a sequence of nested stabbing squares, where only p_3 and p_4 are critical.

LEMMA 7. *There is a constant c depending on the dimensionality and metric of space such that the number of critical squares in Q_0 of side length at least s is $c(s/r)^{1-d}$.*

PROOF. Consider the inclusion tree $T(C)$ consisting of the critical squares of side length at least s. (That is, square p is an ancestor of square q in $T(C)$ iff $p \supseteq q$). We call a critical square a branching node if it has at least two children in $T(C)$, or a non-branching node otherwise. A non-branching node either is a leaf of $T(C)$, or covers more volume of R than its only child node in $T(C)$ does, by the definition of critical squares. Two quadtree squares cover different areas (not necessarily disjoint) of R only if they have different intersections with the surface of R (the inner face of A), so they must also cover different areas of A. Therefore for each non-branching node $p \in T(C)$, there is a unique area of A covered by p but not by any other non-branching nodes of $T(C)$. The volume of this area is $(s^{d-1} \cdot \varepsilon r)/c$ for some constant c determined by the dimensionality and metric of space, since each critical square is a hypercube of side length $2^i s$ for some i. Thus the total number of non-branching nodes in $T(C)$ is $c(s/r)^{1-d}$, since the total volume of A is $O(\varepsilon r^d)$. Therefore $|C| = |T(C)|$ is also $c(s/r)^{1-d}$. □

We also need the following packing lemma for disjoint stabbing squares (an analogy of Lemma 3 in [7]).

LEMMA 8. *There is a constant c depending on the dimensionality and metric of space such that the number of disjoint stabbing squares with side length at least s is $c(s/r)^{1-d}$.*

PROOF. Immediately follow the constant ρ of Q_0. □

5.3 Searching for the Critical Squares

A stabbing square has all its ancestor squares stabbing and a non-stabbing square has all its descendant squares non-stabbing. Therefore the stabbing squares form a pruned tree Q_0' of Q_0. The critical squares further partition the pruned tree Q_0' into equi-stabbing paths such that the squares in the same path cover the same area of $R \cup A$, with the tail (the lowest and smallest square) of the path being a critical square. Here we provide an operation that, given any stabbing square $p \in Q_0$, find the critical square $q \in Q_0$ that covers the same area of $R \cup A$ as p does, i.e., the tail of the equi-stabbing path in Q_0' that p belongs to.

LEMMA 9. *Given R, A, and a compressed quadtree Q_0 with n points, let $p \in Q_0$ be a stabbing square and $q \in Q_0$ be the critical square that covers the same area of $R \cup A$ as p does. Then we can search from p to q in $O(\log n)$ time.*

PROOF. This searching operation is similar to the point location search in Section 3.2 which looks for the smallest square that covers a query location. The major difference is the local comparison rules used by the two searches. In point location search, we start from the root of Q_l, the highest level in the skip structure. At each square $q \in Q_i$ we either go to a child square in Q_i that covers the query location, if such a child square exists, or jump to the next level $q \in Q_{i-1}$. In the critical square search, assuming that p is not critical in Q_0, we promote to Q_j where Q_j is the highest level in which p is not a critical square, then start the search from $p \in Q_j$. At each square $q \in Q_i$ we either go to a child square in Q_i that covers the same area of $R \cup A$ as p does, if such a child square exists, or jump to the next level $q \in Q_{i-1}$. [1]

By the same argument as the ones in Lemma 2 and Lemma 5, the number of searching steps performed at each level Q_i is constant. Therefore the total searching time through all levels is $O(\log n)$ since there are $O(\log n)$ levels in the skip structure. For the randomized skip quadtrees this time bound is not only the expected running time but also happens w.h.p. (as shown in Lemma 2). For the deterministic skip quadtrees this is the worst case running time. □

We conclude with following theorem, and extend the result to more general query regions next without a proof. The extension needs only preliminary geometric knowledge.

THEOREM 10. *We can answer any $(1+\varepsilon)$-approximate range query (v, r, ε) in $O(\log n + \varepsilon^{1-d})$ time.*

PROOF. We start from the root of Q_0 and search through Q_0 to find out all critical squares in Q_0 by using the algorithm in Lemma 9. For each critical square we look for its child squares in Q_0, include or exclude a child square into the range counting if it is an in square or out square. The query is answered when such search is done.

We follow the analytical method of Theorem 1 in [7] to calculate the searching time. For the big critical squares of side length at least r, there are only constant number of such squares by Lemma 7, and searching each of them takes $O(\log n)$ time by Lemma 9. For the small critical squares of side length less than r, the searching time for all of them won't exceed the total number of stabbing squares of side length less than r, which is counted to be $O(\varepsilon^{1-d})$ by using Lemma 8. (See proof of Theorem 1 in [7] for details of this counting.) Again, the result is not only the expected running time but also happens w.h.p. for randomized skip quadtrees, and the worst case running time for deterministic skip quadtrees. □

COROLLARY 11. *We can answer a $(1+\varepsilon)$-approximate range query with convex or non-convex k-fat region in Minkowski space in $O(\log n + \varepsilon^{1-d})$ or $O(\log n + \varepsilon^{-d})$ time, respectively.*

6. APPROXIMATE NN QUERIES

Let the query point (a location) be v and the nearest neighbor (NN) of v in the data set S be x with $dist(v,x) = \min_{z \in S} dist(v,z)$. The $(1+\varepsilon)$-approximate NN query returns a data point $y \in S$ with $dist(v,y) \leq (1+\varepsilon)dist(v,x)$. Together, v and $dist(v,x)$ give a range R (for example, a hyper-sphere for Euclidian distance) which contains no data points, and ε further gives an annulus A. We call them the NN range and the NN annulus, respectively. Therefore one can use the same heuristic for approximate range queries to solve approximate NN queries. The major difference here is that the radius $dist(v,x)$ of R is unknown. This problem of handling range queries with an unknown range can be solved by using a priority queue to expand squares in order by their distance from v. (See Arya et al. [3, 7, 8].) We'll show that this priority queue technique allows skip quadtrees to perform approximate NN queries efficiently.

6.1 Frame of Algorithm

We maintain a minimum priority queue P containing some interesting squares of the compressed quadtree Q_0. The priority of a square p is the distance of p (i.e. from the nearest corner or face) to v. We consider each delete-min of P as a *round*. Here we denote by $MIN(P)$ the square with minimum priority in P and $\min(P)$ the priority of $MIN(P)$. In each round we delete $p = MIN(P)$ from P, however we are not simply splitting p and inserting the children of p in Q_0 back into P. Instead, we search downward Q_0 from p via the skip structure to a descendant of p, say q, and then insert some subsquares of p hanging off the searching path back into P. If a data point instead of a smaller interesting square is found during the search, we consider this point as being visited. And for all visited points we maintain an NN candidate y, that is, the one closest to v. We maintain the following property throughout the algorithm.

- Any point location $u \in S$ that is not covered by any squares in P is at least $(1+\varepsilon)$ times further to v than the NN candidate y. I.e., $\forall\, u \notin P$, $dist(v,u) \geq (1+\varepsilon)dist(v,y)$.

The algorithm stops at any of the following two conditions, or until P becomes empty.

1. If the NN candidate y is close enough to v such that $dist(v,y) \leq (1+\varepsilon)\min(P)$, then output y;
2. If $p = MIN(P)$ is small enough such that $dist'(v,p) \leq (1+\varepsilon)dist(v,p)$, which means that p is an in square with respect to R and A, then output an arbitrary data point in p. Here $dist'(v,p)$ and $dist(v,p)$ respectively denote the distance to v from a furthest corner or face of q, and that from the nearest corner or face of q.

The above property and halting conditions imply that the output is a qualified approximate NN. (See Arya et al. [8] for details.) Now we are left with only two questions: How deep shall we search down from p? And which subsquares of p should be inserted back into P? We give the details next.

6.2 Smallest Equidistant Squares

The interesting squares of Q_0 are partitioned into equivalence classes of *equidistant squares* according to their distance to v. Each such class forms an *equidistant subtree* of Q_0. For the simplicity of algorithm, we further partition the subtree into *equidistant paths* by disconnecting at each branching node of the subtree. That is, the tail (smallest square) of each equidistant path is either closer to v than any of its child squares, or has at least two child squares that are equidistant to v with itself. In each round of the above algorithm we search from p down to the tail of the equidistant path that p belongs to.

We now provide the searching operation that, given any $p \in Q_0$, finds the tail of the equidistant path p belongs to. This is similar to the point location search in Section 3.2 and the critical square search in Section 9, except using another local comparison rule. Here we are given a square p and start from p at the highest level in the skip structure that contains p. At each level Q_i and square $q \in Q_i$, we first check if $q \in Q_0$ has two or more child squares equidistant with p. If so we stop and return q. Otherwise we either go to a child square of q in Q_i that is equidistant to p, if such a child square exists, or jump to the next level $q \in Q_{i-1}$. The correctness and running time follow that for the point location search. [2]

[1]With the computational model in Section 2.2, the area of $R \cup A$ covered by a stabbing square can be determined and compared with that of another stabbing square in constant time.

[2]The distance of a square to the query point is also computable in constant time with the computational model in Section 2.2.

6.3 Stabbing Squares on an Equidistant Path

We now discuss which subsquares hanging off the path between p and q in Q_0 could possibly be stabbing squares (with respect to the NN range and NN annulus). Let's first consider the case that v is closest to a corner of p so that the squares between p and q on the equidistant path go directly towards that corner. (See Fig 7(a) for an illustration in 2D Euclidian space.)

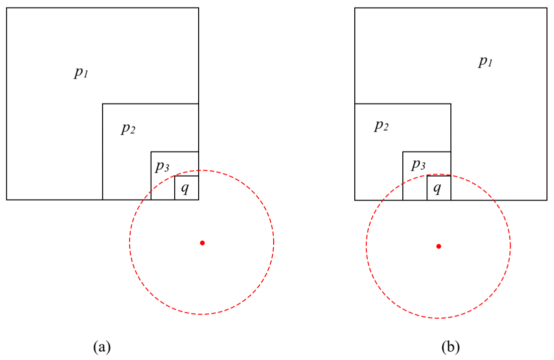

Figure 7: The lowest squares on an equidistant path which are possibly stabbing squares. Circles show the maximum possible $R \cup A$ (not drawn to scale). Although p_1 is impossible to be stabbing in (a), it is possible in (b) since it is the lowest ancestor of q that goes to the right.

LEMMA 12. *There is a constant c depending on the dimensionality and metric of space such that, for any ancestor square q' of q that is more than $\log(c/\varepsilon)$ steps above q in Q_0, if a child of q' hanging off the path between q' and q is a stabbing square, then q is an in square.*

PROOF. Let $s(q)$ be the side length of q. Then a constant c' can be determined by the dimensionality and metric of space such that for any point location u outside q, $dist'(u,q) - dist(u,q) \leq c's(q)$. (In fact, $c's(q)$ is the diameter of q. For example, $c' = \sqrt{2}$ for 2D Euclidian space.) Let q'' be the child of q' that is also an ancestor of q. Since a parent square at least doubles the side length of a child square and q'' is at least $\log(c/\varepsilon)$ steps above q, the side length $s(q'')$ is at least $cs(q)/\varepsilon$. Therefore the distance from any sibling of q'' to v is at least $cs(q)/\varepsilon$. If any sibling of q'' is a stabbing square, then the radius $r(R)$ of the NN range R is at least $cs(q)/\varepsilon$. Let $c = (1+\varepsilon)c'$. Then we have

$$\begin{aligned} dist'(v,q) &\geq r(R) \geq \frac{cs(q)}{\varepsilon} = \frac{(1+\varepsilon)c's(q)}{\varepsilon} \\ &\geq \frac{(1+\varepsilon)(dist'(v,q) - dist(v,q))}{\varepsilon}. \end{aligned}$$

This gives $dist'(v,q) \leq (1+\varepsilon)dist(v,q)$, meaning that q is an in square. □

According to Lemma 12, q and its lowest $\log(c/\varepsilon)$ ancestors are the only possibly stabbing squares on the equidistant path. Therefore we only need to expand these squares and insert back to P the child squares of them that hang off the equidistant path of p. Because other subsquares of p are either not stabbing squares, or q will become an in square, in either case dropping those subsquares doesn't violate the property mentioned in the query algorithm.

Lemma 12 is for a path that goes directly to a corner. However as shown in Fig. 7(b), if the path has bends, then in addition to the $\log(c/\varepsilon)$ lowest ancestors of q, we should also consider for each of the 2^d directions the lowest ancestor of q that goes towards that direction, regardless if this ancestor is within $\log(c/\varepsilon)$ steps above q or not. The total number of possibly stabbing squares is then $2^d + \log(c/\varepsilon) = O(\log \varepsilon^{-1})$. Finding these squares from q can be done in $O(1)$ time per square if we associate additional 2^d pointers to each square in Q_0 pointing to its closest ancestors for each direction. These pointers can also be updated in $O(1)$ time during an insertion or deletion of a point. We conclude with

THEOREM 13. *We can answer a $(1+\varepsilon)$-approximate NN query in $O(\varepsilon^{1-d}(\log n + \log \varepsilon^{-1}))$ time for any distance metric that generates convex equidistant regions (such as Minkowski distance).*

PROOF. The correctness follows the above mentioned property and halting conditions of the algorithm, and the fact that the subsquares we drop during each round don't violate the property.

Now we bound the running time of a query. In each round we perform one delete-min from the priority queue P, one search for the smallest equidistant square, and $O(\log(1/\varepsilon))$ insertions to P. We've already shown that the search of smallest equidistant square takes $O(\log n)$ time. Delete-min from P can be done in $O(\log n)$ time and insertion to P in $O(1)$ time with standard priority queue implementations. Therefore each round takes $O(\log n + \log \varepsilon^{-1})$ time. It remains to count the number of rounds. Each round ends up with a smallest equidistant square which is either a critical square or an in square. The number of critical squares is bounded to $O(\varepsilon^{1-d})$ by Lemma 7 since the side length of a critical square is at least $c\varepsilon r$ for some constant c. The number of in squares can be bounded to the same upper bound by similar argument to the one in the proof of Lemma 7, or more straightforwardly (if we don't care the constant factor of 2^d), to 2^d times of the number of critical squares since the parent of each in square is a critical square. Therefore the total number of rounds performed before halting is $O(\varepsilon^{1-d})$. □

7. CONCLUSION

We've presented a dynamic data structure, the skip quadtree (or skip octree), for multidimensional data. In addition to providing efficient operations of inserting and deleting a point in $O(\log n)$ time, we demonstrate the power of this data structure for efficiently answering the following three types of geometric queries: point location search in $O(\log n)$ time, $(1+\varepsilon)$-approximate range counting in $O(\varepsilon^{1-d} + \log n)$ time, and $(1+\varepsilon)$-approximate nearest neighbor searching in $O(\varepsilon^{1-d}(\log n + \log \varepsilon^{-1}))$ time. These query times match the best previous results (e.g., in [7, 8, 17, 19]) achieved on the static data structures such as BBD trees and BAR trees. [3]

Not only is the skip quadtree a fully dynamic data structure, it is also significantly simpler to implement than the static BBD trees and BAR trees. We've provided both randomized and deterministic versions of skip quadtrees, with the above time bounds being expected bounds as well as bounds with high probability for the randomized version and worst-case bounds for the deterministic version. Furthermore, both versions of skip quadtrees take linear space.

Acknowledgements

The authors would like to thank David M. Mount, Hanan Samet, Ke Yi, and two anonymous referees for their valuable comments.

[3]For approximate NN queries, the best previous result is $O(\varepsilon^{-d} \log n)$ time in [8]. However, we think this bound could be characterized as $O(\varepsilon^{1-d} \log n)$ if restricted to Minkowski distance, considering the fact that Minkowski distance generates convex equidistant regions. Under a reasonable assumption of $\varepsilon^{-1} = O(n^c)$ for some constant c, skip quadtrees also match this bound in answering such queries.

8. REFERENCES

[1] S. Aluru. Quadtrees and octrees. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 19–1–19–26. Chapman & Hall/CRC, 2005.

[2] S. Aluru and F. E. Sevilgen. Dynamic compressed hyperoctrees with application to the N-body problem. In *Proc. 19th Conf. Found. Softw. Tech. Theoret. Comput. Sci.*, volume 1738 of *Lecture Notes Comput. Sci.*, pages 21–33. Springer-Verlag, 1999.

[3] S. Arya, T. Malamatos, and D. Mount. Space-efficient approximate Voronoi diagrams. In *Proc. 34th Annual ACM Sympos. Theory Comput.*, pages 721–730, 2002.

[4] S. Arya, T. Malamatos, and D. M. Mount. Space-time tradeoff for approximate spherical range counting. In *16th ACM-SIAM Annual Symposium on Discrete Algorithms, (SODA05)*, pages 535–544, 2005.

[5] S. Arya and D. Mount. Computational geometry: Proximity and location. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 63–1–63–22. Chapman & Hall/CRC, 2005.

[6] 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.

[7] S. Arya and D. M. Mount. Approximate range searching. *Comput. Geom. Theory Appl.*, 17:135–152, 2000.

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

[9] T. Asano, M. Edahiro, H. Imai, M. Iri, and K. Murota. Practical use of bucketing techniques in computational geometry. In G. T. Toussaint, editor, *Computational Geometry*, pages 153–195. North-Holland, Amsterdam, Netherlands, 1985.

[10] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

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

[12] 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 Comput. Sci.*, pages 188–199. Springer-Verlag, 1993.

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

[14] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[15] L. Devroye, J. Jabbour, and C. Zamora-Cura. Squarish ık-ıd trees. *SIAM J. Comput.*, 30(5):1678–1700, 2000.

[16] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 9th ACM STOC*, pages 365–372, 1987.

[17] C. A. Duncan. *Balanced Aspect Ratio Trees*. Ph.D. thesis, Department of Computer Science, Johns Hopkins University, Baltimore, Maryland, 1999.

[18] C. A. Duncan and M. T. Goodrich. Approximate geometric query structures. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 26–1–26–17. Chapman & Hall/CRC, 2005.

[19] C. A. Duncan, M. T. Goodrich, and S. Kobourov. Balanced aspect ratio trees: combining the advantages of k-d trees and octrees. *J. Algorithms*, 38:303–333, 2001.

[20] A. Efrat, M. J. Katz, F. Nielsen, and M. Sharir. Dynamic data structures for fat objects and their applications. *Comput. Geom. Theory Appl.*, 15:215–227, 2000.

[21] A. Efrat, G. Rote, and M. Sharir. On the union of fat wedges and separating a collection of segments by a line. *Comput. Geom. Theory Appl.*, 3:277–288, 1993.

[22] K. Fujimura, H. Toriya, K. Tamaguchi, and T. L. Kunii. Octree algorithms for solid modeling. In *Proc. Intergraphics '83*, volume B2-1, pages 1–15, 1983.

[23] E. N. Hanson and T. Johnson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Workshop on Algorithms and Data Structures (WADS)*, pages 153–164, 1991.

[24] D. T. Lee. Interval, segment, range, and priority search trees. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 18–1–18–21. Chapman & Hall/CRC, 2005.

[25] S. Leutenegger and M. A. Lopez. R-trees. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 21–1–21–23. Chapman & Hall/CRC, 2005.

[26] M. A. Lopez and B. G. Nickerson. Analysis of half-space range search using the *k*-d search skip list. In *14th Canadian Conference on Computational Geometry*, pages 58–62, 2002.

[27] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proc. Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 367–375, 1992.

[28] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 367 – 375, 1992.

[29] B. F. Naylor. Binary space partitioning trees. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 20–1–20–19. Chapman & Hall/CRC, 2005.

[30] B. G. Nickerson. Skip list data structures for multidimensional data. Technical Report CS-TR-3262, 1994.

[31] J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 725–764. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[32] J. A. Orenstein. Multidimensional tries used for associative searching. *Inform. Process. Lett.*, 13:150–157, 1982.

[33] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[34] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

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

[36] H. Samet. Spatial data structures. In W. Kim, editor, *Modern Database Systems, The Object Model, Interoperability and Beyond*, pages 361–385. ACM Press and Addison-Wesley, 1995.

[37] H. Samet. Multidimensional data structures. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, pages 18–1–18–28. CRC Press, 1999.

[38] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, CA, 2005.

[39] H. Samet. Multidimensional spatial data structures. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 16–1–16–29. Chapman & Hall/CRC, 2005.

APPENDIX

A.

Here we give an approximation of the progressions in (1) in the proof of Lemma 2. If $f(x) \geq 0$ is a monotone decreasing function for $x \geq i$, then the progression of $f(x)$ can be approximated by its integral as following:

$$\sum_{x=1}^{i-1} f(x) + \int_{x=i}^{\infty} f(x)dx \leq \sum_{x=1}^{\infty} f(x) \leq \sum_{x=1}^{i} f(x) + \int_{x=i}^{\infty} f(x)dx.$$

By setting sufficiently big i (bigger i gives better accuracy) and calculating the following integrals

$$\int_{x=i}^{\infty} \frac{x^2}{2^x}dx = \frac{1}{2^x \ln^3 2}[(\ln 2 \cdot x)^2 + 2\ln 2 \cdot x + 2] \mid_{x=i}$$

and

$$\int_{x=i}^{\infty} \frac{x}{2^x}dx = \frac{1}{2^x \ln^2 2}(\ln 2 \cdot x + 1) \mid_{x=i},$$

we can bound the two progressions in (1) as follows.

$$\sum_{x=1}^{\infty} \frac{x^2}{2^x} \leq 6.005, \quad \text{taking } i = 14;$$

$$\sum_{x=1}^{\infty} \frac{x}{2^x} \leq 2.005, \quad \text{taking } i = 10.$$

B.

Here we show, for the proof of Lemma 2, that the estimation of $E(j) = 2$ for the local structure of Q_i with $p_0, p_1, \cdots, p_m$ each having two non-empty quadrants and each non-empty quadrant hanging off the path containing one point is the worst possible. Assume that the adversary is given one more point to add to the structure and he wants to use this point efficiently to increase $E(j)$ the most. There are two ways to increase $Pr(j)$ for a certain j as following.

One is to add a point to the non-empty quadrant of p_{m-j} hanging off the path, which increases the probability of this quadrant being non-empty in S_{i+1}. The other is to add one more non-empty quadrant to p_{m-j}, which makes the case that none of the points in p_{m-j+1} is selected also contribute to $Pr(j)$. The former way is more efficient and it increases $Pr(j)$ from $\frac{1}{2} \cdot \frac{j+1}{2^{j+1}}$ to $\frac{3}{4} \cdot \frac{j+1}{2^{j+1}}$, a 0.5 time increase to $jPr(j)$. However, either way of adding a point to p_{m-j} will inevitably decrease $Pr(j')$ for all $j' \geq j+1$. In proof of Lemma 2 and Appendix A we bound $\sum_{x=j+1}^{\infty} xPr(x)$ as

$$\begin{aligned} \textstyle\sum_{x=j+1}^{\infty} xPr(x) &\leq \tfrac{1}{4}(\textstyle\int_{x=j+1}^{\infty} \tfrac{x^2}{2^x} d(x) + \int_{x=j+1}^{\infty} \tfrac{x}{2^x} d(x)) \\ &\sim \tfrac{1}{4} \cdot \tfrac{j^2}{2^j \ln 2} \sim \tfrac{jPr(j)}{\ln 2} = 1.44jPr(j). \end{aligned}$$

Adding one point to p_{m-j} decreases $Pr(j')$ for any $j' \geq j+1$ from $\frac{1}{2} \cdot \frac{j'+1}{2^{j'+1}}$ to $\frac{1}{2} \cdot \frac{j'+2}{2^{j'+2}}$ so that it decreases the above bound from $\sim 1.44jPr(j)$ to $\sim 0.72jPr(j)$. Therefore the over all decrease of $E(j)$ is more than enough to kill the increase. It's also clear that adding more points will be less efficient than adding the first one for the adversary.

Notice that this argument directs to the bound $E(j) \leq 2$ but not to the structure itself. For a finite path of squares (so the bound is not tight), one can always add one point to the quadrant hanging off the highest square to increase $E(j)$ a little, that is, from $2-\delta$ to $2-\delta'$ for some $\delta > \delta' > 0$.

C.

Here we use Chernoff bounds to do the high probability analyses in proofs of Lemma 2 and Theorem 3. For Lemma 2, let X be the number of coin flips in order to get $2c\log n$ heads. We already know that $E(X) = c'\log n$ for some constant c and we set $\delta \geq 2e-1$. Therefore

$$\begin{aligned} &Pr[X > (1+\delta)c'\log n] \\ &< [\tfrac{e^\delta}{(1+\delta)^{(1+\delta)}}]^{c'\log n} < [\tfrac{e^\delta}{(2e)^{(1+\delta)}}]^{c'\log n} < (\tfrac{1}{2^{1+\delta}})^{c'\log n} = \tfrac{1}{n^{(1+\delta)c'}}. \end{aligned}$$

This shows that $X = O(\log n)$ w.h.p.

Similarly, for Theorem 3, let X be the number of heads in a sequence of coin flips before seeing the first tail. We know that $E(X) = 1$ and we set $\delta = c\log n - 1 \geq 2e-1$. Therefore

$$\begin{aligned} &Pr[X > 1+\delta = c\log n] \\ &< \tfrac{e^\delta}{(1+\delta)^{(1+\delta)}} < \tfrac{e^\delta}{(2e)^{(1+\delta)}} < \tfrac{1}{2^{1+\delta}} = \tfrac{1}{2^{c\log n}} = \tfrac{1}{n^c}. \end{aligned}$$

This shows that $X = O(\log n)$ w.h.p.