A Succinct, Dynamic Data Structure for Proximity Queries on Point Sets

Prayaag Venkat* David M. Mount†

Abstract

A data structure is said to be *succinct* if it uses an amount of space that is close to the information-theoretic lower bound, but still allows for efficient query processing. Quadtrees are among the most widely used data structures for answering queries on point sets in Euclidean space. In this paper we present a succinct quadtree structure. Our data structure can efficiently answer approximate range queries and approximate nearest neighbor queries and supports insertion and deletion of points.

1 Introduction

In the field of computational geometry, there exists a wide variety of data structures that are used to efficiently solve retrieval problems on point sets. A set P of n points is given in real Euclidean space $\mathbb{R}^d$, which is preprocessed into a data structure. In *range searching*, a query region Q is provided and the points of $P \cap Q$ are to be reported or counted. In *nearest neighbor searching*, a query point q is given, and the nearest point of P to q is to be reported. If some degree of error is allowed, it is often possible obtain significantly more efficient solutions.

A number of data structures have been proposed for approximate retrieval problems for points sets. We will focus on methods based on regular subdivisions of space, and in particular on the quadtree and its variants (see, e.g., [9, 17]). An important feature of the quadtree that makes it appropriate for approximate retrieval problems, is that it decomposes space into disjoint regions, which are of constant combinatorial complexity and bounded aspect ratio.

*Department of Computer Science, University of Maryland College Park, pkvasv@gmail.com

†Department of Computer Science, University of Maryland College Park, mount@cs.umd.edu. Supported by NSF grant CCF-1117259 and ONR grant N00014-08-1-1015.

The focus of this paper will be on dynamic, space-efficient data structures for answering approximate range and approximate nearest-neighbor queries. Although traditional implementations of quadtrees provide fast query times and use $O(n)$ words of space, when dealing with very large data sets it is often desirable to have even higher standards for space efficiency. A data structure is said to be *succinct* if the number of bits needed to represent the structure is close to the information-theoretic minimum number of bits needed to store the structure. More formally, if Z denotes this information-theoretic lower bound, then a succinct data structures uses only $Z + o(Z)$ bits.

In the literature there are several examples of succinct representations of trees that provide efficient support for a diverse set of operations. The first of these representations was introduced by Jacobson [11]. There has been a good deal of research since then focused on supporting different types of structures and a wider variety of operations (see, e.g., [4, 14–16]). Arroyuelo *et al.* have shown that succinct tree implementations are quite efficient in practice [1].

Succinct data structures have also been applied to geometric data structures. Bose *et al.* [5] showed how to answer orthogonal range queries succinctly. Hudson [10] applied principles from succinct data structures to the quadtree and used lossy compression techniques to represent a set of n well-spaced points using only $O(n)$ bits (irrespective of the number of bits needed to represent the point coordinates). Perhaps the ultimate in succinctness is Chan's "minimalist" in-place randomized algorithm for approximate nearest neighbor searching, which stores nothing more than the points sorted in Morton order (defined below) [6]. Although Chan's structure uses the minimum number of bits, it cannot answer range counting queries.

Here we are interested in developing a succinct representation of a quadtree for storing the points of P. We assume the word RAM model of computa-

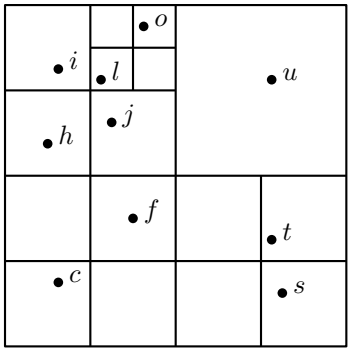
tion, in which memory consists of $w = \Theta(\log n)$ bit words. We assume that each point is represented as a d-element vector of coordinates, where each coordinate is represented as a bit string of length b. We make the relatively weak assumption that b is polynomial in w. In order to answer range queries (even approximately) it is necessary to represent each point to its full precision, and therefore $d \cdot b \cdot n$ is a lower bound on the number of bits needed by any data structure. However, given the quadtree's structure, it is possible to infer information about the coordinates to reduce the space requirements. For range counting, we assume that point weights are drawn from a group (thus allowing subtraction), and the maximum weight is polynomial in w.

2 Preliminaries

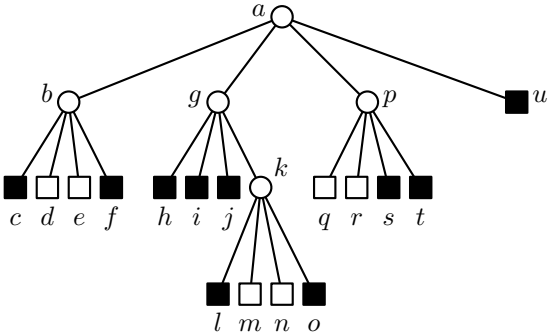
2.1 Basic Definitions and Results

Recall that we have an n-element point set P in $\mathbb{R}^d$, and we assume that each coordinate is represented as a bit string of length b. We are interested in answering approximate nearest neighbor and approximate range queries over this set. In *ε-approximate nearest neighbor searching* we are given a query point q and an error parameter $\varepsilon > 0$. The query algorithm may return any point $p \in P$ whose distance to q is at most $(1+\varepsilon)\delta$, where δ is the distance from q to its closest point in P. In *ε-approximate range searching* we are given two convex shapes, an inner range Q^- and an outer range Q^+, where $Q^- \subset Q^+$, and the boundaries of these two ranges are separated by a distance of at least $\varepsilon \cdot \text{diam}(Q^-)$. We wish to count (or report) any subset of P that includes all the points of P lying within Q^- and may optionally include points lying in $Q^+ \setminus Q^-$. In general, we allow each point to be associated with an integer weight (all having an equal number of bits), and a counting query returns the weighted sum of such a subset. The query shapes are assumed to satisfy the *unit-cost assumption*, which states that it is possible to determine in constant time whether a hypercube lies entirely inside Q^+ or entirely outside Q^-.

Our data structure is a succinct representation of a standard (uncompressed) *quadtree* (formally a PR-quadtree [17]). We assume that the points of the set P have been transformed to lie within the half-open unit hypercube $[0,1)^d$. Each node of the quadtree represents an axis-aligned hypercube



(a) A quadtree decomposition of a point set. Each point is labeled with its corresponding quadtree leaf node (see (b)).



(b) A quadtree for the point set of (a), where white circles denote internal nodes, black squares denote leaf nodes that contain a point, and white squares denote empty leaf nodes.

Figure 1

in $\mathbb{R}^d$, called a *quadtree box*, and is associated with the points of P that lie within its box (see Figure 1). Each level of the tree corresponds to one bit of each coordinate, and therefore the tree's depth cannot exceed b. Let $\Phi = \Phi(P)$ denote the *aspect ratio* (or *spread*) of the point set, defined to be the ratio between the maximum and minimum inter-point distances. It is also well known that the tree's depth is $O(\log \Phi)$ (see, e.g., [9]). We now present our main results.

Theorem 1 *Consider an n-element point set P in $\mathbb{R}^d$, where each coordinate is represented as a bit string of length b. Let Φ denote P's aspect ratio. Let m denote the total number of nodes of an (uncompressed) quadtree $T(P)$ defined by P. There exists a representation of P and $T(P)$ that uses $(d+2)m + o(m)$ bits for $T(P)$ and (conditioned on $T(P)$) a minimum number of bits to represent P*

that supports the following operations:

(*i*) *ε-approximate range counting queries can be answered in $O((\log\Phi + 1/\varepsilon^d)\lg^2\lg m)$ time.*

(*ii*) *ε-approximate nearest neighbor and ε-approximate k-nearest neighbors queries can be answered in $O((1/\varepsilon)^d(\log\Phi)\lg^2\lg m)$ and $O(((1/\varepsilon)^d + k)(\log\Phi)\lg^2\lg m)$ time, respectively, for any fixed k.*

Recall that for range counting we assume that point weights are drawn from a group, and the maximum weight is polynomial in w. We can generalize this to dynamic point sets, where insertion and deletions are supported. In this context, let Φ be an upper bound on the aspect ratio of the point set at any time.

Theorem 2 *There is a dynamic version of the data structure of Theorem 1 that uses the same asymptotic space and supports the following operations, where $N = d \cdot b \cdot n$ denotes the total number of coordinate bits.*

(*i*) *Points can be inserted in $O\big(\log\Phi + b\frac{\log N}{\log\log N} + \log m\big)$ amortized time.*

(*ii*) *Points can be deleted in $O\big(\big(b\frac{\log N}{\log\log N} + \log m\big)\log\Phi\big)$ amortized time.*

(*iii*) *The query times of Theorem 1 are slower by a multiplicative factor of $O\big(\frac{\log N}{\log\log N}\big)$.*

2.2 The Morton Code

An important concept underlying our approach is the Morton order of a set of points. The *Morton order* (or *Z-order*) is a mapping from the unit hypercube $[0,1)^d$ to one dimensional space [13]. Consider a point $p \in [0,1)^d$, which we assume to be presented as a d-element vector of coordinates $p = (p_1, \ldots, p_d)$, where each p_i is a b-element bit string in base-2, that is, $p_i = 0.p_{i1}p_{i2}\ldots p_{ib}$. The *Morton code* of p, denoted $M(p)$, is obtained by interleaving these bits into a string of length $d \cdot b$

$$M(p) \;=\; p_{11}\ldots p_{d1}\; p_{12}\ldots p_{d2}\;\cdots\; p_{1b}\ldots p_{db}.$$

We can associate each quadtree node u with a Morton code, denoted $M(u)$, as follows. Let $\ell(u)$ denote u's level in the tree. $M(u)$ is the Morton code of the lower left corner of u's quadtree box using coordinate bit strings of length ℓ (so that the Morton code is of total length $d \cdot \ell$ bits). The following are easy consequences:

- The side length of u's box is $1/2^\ell$.
- The Morton codes of u's 2^d children in a quadtree subdivision are $M(u) + \omega$, where ω ranges over all bit strings of length d and "+" denotes concatenation.
- A point p lies within u's quadtree box if and only if $M(u)$ is a prefix of $M(p)$.

2.3 Review of Succinct Trees

The minimum number of bits required to represent an element from a set S is $\lg|S|$. We will represent a quadtree in $\mathbb{R}^d$ as a rooted *cardinal k-ary tree*, for $k = 2^d$. This means that every node has k positions that may or may not contain a pointer to a child node. It is well known that the number of such trees of degree k is $\frac{1}{kn+1}\binom{kn+1}{n}$ [8], so the information-theoretic lower bound on the number of bits needed to represent an n-node cardinal tree of degree k is [4, 7] $\lg\left(\frac{1}{kn+1}\binom{kn+1}{n}\right) \approx 2n + n\lg k - o(n + \lg k)$.

We extend the methods presented by Benoit *et al.* [4] and Davoodi and Rao [7] for succinctly representing quadtrees as k-ary cardinal trees. We begin by defining an *ordinal k-ary tree* to be a rooted tree in which every node has up to k children, but information is not maintained about child nodes that may be absent. The general approach to representing a cardinal tree succinctly is to implement a succinct ordinal tree and then maintain additional information about the children of each node.

The *unary degree sequence* (or *UDS*) of a node in an ordinal tree is defined as follows: for each child that the node possesses, write down a 1-bit (or "(") and after accounting for all the children, write down a single 0-bit (or ")"). To represent the whole tree, the tree is traversed in depth-first order, and the UDS of each node is written down. After obtaining the final bit string, a single 1-bit is added to the front so that every 1-bit has a matching 0-bit.

This representation of tree is called the *depth-first unary degree sequence* (or *DFUDS*) representation (see Figure 2). In the DFUDS representation of an ordinal tree, a node is defined by the index of the first bit of its UDS. Given an ordinal tree, we can use it to represent a cardinal tree by storing information about child nodes (including information pertinent to the application) in a separate array.

Now that we have a succinct representation of the cardinal tree, we introduce basic operations on the DFUDS representation that will form the basis of all other operations. (Proofs and details on

DFUDS Representation $1\,\underset{a}{11110}\ \underset{b}{11110}\ \underset{c}{0}\ \underset{d}{0}\ \underset{e}{0}\ \underset{f}{0}\ \underset{g}{11110}\ \underset{h}{0}\ \underset{i}{0}\ \underset{j}{0}\ \underset{k}{11110}\ \underset{l}{0}\ \underset{m}{0}\ \underset{n}{0}\ \underset{o}{0}\ \underset{p}{11110}\ \underset{q}{0}\ \underset{r}{0}\ \underset{s}{0}\ \underset{t}{0}\ \underset{u}{0}$

Figure 2: The DFUDS representation of the quadtree from Figure 1b.

their actual implementation and requirements can be found in [1, 4, 11, 16].) Essentially, the DFUDS representation of a tree is a bit string that can support primitive operations with the help of auxiliary data structures. Given a DFUDS bit string D, $c \in \{0, 1\}$, and i ranging over the positions in D, the following constant time operations are supported and require only $o(n)$ bits.

- $rank_c(i)$ returns the number of occurrences of c in the DFUDS bit string D up to position i.
- $select_c(i)$ returns the position of the ith occurrence of c in D.

3 Succinct Quadtree

We now adapt this succinct tree structure to the special case of the quadtree, and we discuss how to store the associated point set. The total space requirements of the point set are $d \cdot b \cdot n$ bits. We will exploit the fact that the quadtree structure itself provides partial information as to where the points reside.

Consider an internal node u at level ℓ of the tree. For each child v of u, the d bits of the corresponding index are equal to the next d-bits of the Morton code of the points that lie within v's quadtree box. (The quadtree is effectively a trie defined by the Morton codes of P's points.) These are the $(\ell+1)$st bits of each of the d coordinates of these points. Therefore, as we traverse a path of length ℓ in the tree from the root to a leaf, we implicitly know the first ℓ bits of each coordinate of any point that lies within this leaf.

As mentioned earlier, we represent the quadtree as a k-ary rooted cardinal tree, for $k = 2^d$. The children of a node are indexed from 0 to $2^d - 1$. Following Benoit *et al.* [4], for each internal node we store only the children that contain at least one point of P. Based on the implementation described by Davoodi and Rao [7], our tree is represented as a DFUDS bit string, which we denote by $D(P)$ or simply D when P is clear. Our subsequent processing will make use of the following operations, which run in $O(1)$ time.

- Given a node at position i in the DFUDS bit string D, compute the position of its jth child, where $0 \le j < 2^d$ (see [4] for the proof).

- As we traverse a path from the root to any node, whenever we visit a node at level ℓ, we can maintain the first ℓ bits of the coordinates of the points that lie within the quadtree box associated with this point.

As mentioned above, once we know the leaf node containing a point at some level ℓ, we know the first ℓ bits of each of the coordinates of this point. Therefore, the minimum amount of information needed to faithfully represent the point's exact position are the remaining $b - \ell$ bits of each coordinate. We call these the *leftover coordinate bits*. We store these leftover bits for all the points contiguously in a single bit string. These are sorted first according to a depth-first ordering of the leaves of the tree (which is the same as the Morton ordering of P). For each point, we store d bit strings, one for each of the point's coordinates. Therefore, if a point is stored at a leaf at level ℓ of the tree, this point contributes $d(b - \ell)$ bits to this bit string.

The challenge arising from this representation is locating the leftover bits for a given point p. It suffices to determine how many leftover bits there are in all the points that precede p in the Morton order. Given p, define $P_{<p}$ to be the subset of P that precedes p in Morton order. For any $q \in P$, let $\ell(q)$ denote the level of the leaf node that contains q. Then the starting position of p's leftover bits in this string is $pos(p) = \sum_{q \in P_{<p}} d(b - \ell(q))$.

Clearly d and b are known, and $|P_{<p}|$ can be computed in $O(1)$ time by the standard DFUDS operation *leaf-rank*. (It is straightforward to adapt the algorithm for *leaf-rank* presented in Sadakane and Navarro [16] from ordinal trees to cardinal trees.) Given the index i of a leaf node in the DFUDS string representing a point p, define *leaf-sum*$(i) = \sum_{q \in P_{<p}} \ell(q)$. In the next section we will show how to compute *leaf-sum*(i), the only unknown in the formula above, in $O(\lg^2 \lg m)$ time.

3.1 Leaf sum

In this section we discuss how to answer the *leaf-sum* query defined in the previous section. Following standard methods, our approach is to subdivide the DFUDS bit string into successively smaller blocks such that the query can be answered by accumulating answers from these blocks [11, 14].

First, we divide D into *blocks* of size $t = \lceil \lg^2 m \rceil$ bits each. At the head of each block, we store the sum of the heights of all leaf nodes that precede this block in DFS order. There are $O(m/t)$ such blocks. The maximum *leaf-sum*(i) value that can be stored in a block is at most m^2, because the height of the quadtree cannot exceed m and the number of leaves up to any block is less than m. So, the total space in bits is $O\big(\frac{m}{t} \lg(m^2)\big) = O\big(\frac{m}{\lg m}\big) = o(m)$.

Next, we divide D again into *miniblocks*, of size $s = \lceil \lg^2 \lg m \rceil$ bits each. In each miniblock, we store the sum of the heights of all leaves that precede the miniblock in DFS order but are still in the same block. There are $O(m/s)$ miniblocks altogether, and the maximum *leaf-sum*(i) value stored in any miniblock is $O(t \cdot b)$ because there are no more than t nodes in a block, and the height of the tree is at most b. Also, the tree has at least as many nodes as there are points in P, implying that $n \leq m$. By our assumptions that $b = O(w^c)$ and $w = O(\log n)$, the value stored in any miniblock is at most $O(t \cdot \lg^c m)$. Therefore, the total number of bits for all the miniblocks is $O\left(\frac{m}{s} \lg(t \cdot \lg^c m)\right) = o(m)$.

Finally, we must be able to answer *leaf-sum*(i) queries within a miniblock. Since the result is not uniquely determined by the miniblock's contents, we explicitly calculate the sum of the heights of the leaves that precede the current node. There are at most s nodes within a miniblock, so calculating the heights of each leaf and adding them takes $O(\lg^2 \lg m)$ time.

In conclusion, we compute *leaf-sum*(i) as follows. First, we obtain the value stored at the head of the block that immediately precedes i. Next, we add to this the value stored at the head of the miniblock that immediately precedes i. Finally, we calculate the answer to the intra-miniblock query, add it to the current sum, and return the sum as the final value. It follows that *leaf-sum* queries can be answered in $O(\lg^2 \lg m)$ time.

3.2 Succinctness

First, we show that the quadtree representation is succinct. Recall from Section 2.3 that the minimum number of bits required to represent a k-ary cardinal tree is at least $2m + m \lg k - o(m + \lg k)$, which in the case of quadtrees ($k = 2^d$) is $m(d+2) - o(m + \lg k)$. Following the analysis of Davoodi and Rao [7], and the fact that our auxiliary structure for answering *leaf-sum* queries involves $o(m)$ space, we have the following.

Lemma 3 *Our DFUDS representation stores an m-node quadtree using $(d+2)m + o(m)$ bits.*

In addition to storing the quadtree, we also store the leftover bit coordinates of the points. We assert that conditioned on the existence of the quadtree, these bits must all be stored in order for the data structure to function properly. In particular, given any point $p \in P$, consider a range query consisting of any closed ball centered at p whose radius is smaller than $1/2^b(1+\varepsilon)$. If our representation fails to represent p's position with complete fidelity, this query cannot be correctly answered (even approximately). Also, observe that all information about the point's location that could be gleaned from an inspection of the data structure has been eliminated from the leftover bits. Therefore, given the quadtree structure, the number of bits used to store the leftover coordinates is the minimum required by an information-theoretic argument.

4 Approximate Range Searching

In order to answer ε-approximate range counting queries for P using the succinct quadtree, we adapt the algorithm of Arya and Mount [2]. The algorithm begins at the root of the tree and traverses downwards until there are no more nodes that need to be checked, at which point the algorithm terminates and the total weight is returned. If the current node's box falls completely inside $Q+$, we add the weight of that node to the total weight. If the current node's box falls completely outside of Q^-, we ignore it. If the node is a leaf, then we check whether its point lies within Q^-, and if so, we add its weight to the total. Finally, if the node intersects Q^- but is not entirely contained in Q^+, we apply the process recursively to its children.

First, we need to be able to compare the quadtree box of the current node to the query ranges. Recall

that as we descend the tree, we can maintain the coordinates of the lower left corner of the quadtree box associated with the current node. Since we can also maintain the level in the tree, we can compute its side lengths as well. Given this, we can compare the quadtree box with the ranges in $O(1)$ time. Second, we must be able to access any given child node of the current node in $O(1)$ time. This is handled by the underlying DFUDS-based tree representation (see, e.g., Benoit *et al.* [4]).

Finally, we must return, report, and access the weights of the points. This operation is more complicated and will be described in the next section. If we can perform these operations in the mentioned times, then it follows that the running time is at most the count of the number of nodes visited multiplied by the time it takes to access a point. In the quadtree, similar to the BBD-tree, the number of nodes that must be evaluated is $O(\lg^2 \lg m(\log \Phi + 1/\varepsilon^d))$, as shown by Arya and Mount [2].

4.1 Processing Weights for Range Searching

The weight of a node is the sum of weights of the points descended from this node. Define *weight*(i) to be the weight of the node at the ith position. We present auxiliary data structures, similar to those in Section 3.1, that allow us to compute *weight*(i) in $O(1)$ time using $o(m)$ bits of space. In addition, we require that maximum weight value W is polynomial in the word size w.

First, we divide the DFUDS bit string D into blocks of size $t = \lg^2 m$ bits each. At the head of each block, we store the sum of the weights of all leaves up to that point. There are $O(m/(\lg^2 m))$ blocks, and the maximum weight value stored at any block is nW, so the total number of bits required to store answers at each block is $O(m(\lg(nW))/(\lg^2 m))$. Because W is polynomial in w and the number of points n is not greater than the number of nodes m, this is $o(m)$.

Next, we divide D into miniblocks of size $s = \frac{1}{2}\lg m$ its each. At the head of each miniblock, we store the sum of the weights of all leaves that precede i within its block. There are $O(m/(\lg m))$ miniblocks and the maximum weight value stored at any miniblock is $W \lg^2 m$, so the total number of bits required to store answers at each miniblock is $O(m(\lg(W \lg^2 m))/(\lg m))$, which is $o(m)$ because W is polynomial in w.

Finally, we use a look-up table to store the sum of all leaf weights that precede a given node within a miniblock. There are $2^{\frac{1}{2}\lg m} = \sqrt{m}$ possible distinct miniblocks and there are $O(\lg m)$ nodes to query in a miniblock. The maximum weight value within a miniblock is $O(W \lg m)$, so it costs $O(W\sqrt{m}\lg^2 m) = o(m)$ bits to store a look-up table for intra-miniblock queries.

Using these auxiliary data structures, we can find the sum of the weights of all leaves that precede node i (called *preweight*(i)) by adding up the value stored at the block before i, the value stored at the miniblock before i, and the value stored in the look-up table for i. To find the sum of the weights of all points that fall inside i, we need to find the sum of all weights of leaves that are descendants of i. This is accomplished by finding the *preweight* of the node that immediately follows rightmost leaf of i and subtracting from it *preweight*(i). Thus we have, *weight*(i) =

$$\textit{preweight}(\textit{rightmost-leaf}(i) + 1) - \textit{preweight}(i).$$

It takes constant time to compute this value because there are three $O(1)$ time look-ups per *preweight*, arithmetic, and the *rightmost-leaf* operation which takes $O(1)$ [16].

4.2 Updating Point Weights for Range Search

In the dynamic case, we may wish to insert new points or modify the weight of an existing point. In order to update the weights of the points, we use a solution similar to that of Section 6.4, where we will provide more detail. The basic idea is that for inner miniblock queries, we store look-up tables for miniblocks of all sizes up to and including $\frac{1}{2}\log m$, in the case that a miniblock changes size. Then, we directly modify the values stored at the head of each miniblock that follow the newly inserted node, which takes $O(\log m)$ time because there are at most $O(\log m)$ such miniblocks. Finally, we create an *update array* U that has an element for every block. If a weight is inserted (or deleted), we add (or subtract) the new weight value to the element in the U that corresponds to the block in which the corresponding node is found. When we wish to return a value stored at the head of block i, we obtain the value stored at the head of block i, we find the sum of all values in the U up to an including position i, and return the sum of the preceding two values. If we maintain the U as a dynamic partial

sums array [7], then we can perform all these operations in constant time and $o(m)$ bits of space. In total, updating a point weight takes $O(\log m)$ time.

5 Approximate Nearest Neighbor Searching

In order to answer ε-approximate nearest neighbor queries, we modify the algorithm presented by Arya *et al.* [3]. We now provide a short description here, with details to follow. The basic idea behind the algorithm is to maintain a priority queue of tree nodes in order of increasing distance from the query point to find its approximate nearest neighbor. We show that the priority queue never becomes too large, allowing us to represent it using $o(n)$ bits and answer the query with a slightly modified version of the original algorithm.

In order to answer ε-approximate nearest neighbor queries, we modify the algorithm presented by Arya *et al.* [3]. The algorithm uses the priority search technique in which the basic idea is to search for p' in increasing distance away from q. Arya *et al.* make use of a BBD-tree data structure, but the quadtree is similar in nature and is easily adaptable to the algorithm. First, we maintain a priority queue that contains quadtree nodes that are ordered in increasing distance away from q. Initially, the root of the tree is inserted into the queue. Then, the node v with the highest priority, or closest distance to q, is extracted from the queue. We traverse v to a leaf of the tree by moving to the closest node to q at each step of the traversal. When we reach a leaf node, we repeat the process and extract the next highest priority node from the queue. As we traverse the tree, the siblings of nodes along the path are inserted into the queue. During this process, we keep track of the closest point found inside a node encountered so far.

Arya *et al.* show that at termination, the closest point so far is in fact p'. Furthermore, the number of leaf nodes visited before termination is given by $C_{d,\varepsilon} = O(1/\varepsilon^d)$. We omit the proof, but the basic idea is that p' must be contained in some Euclidean ball centered at q, and the number of leaf nodes that can intersect that ball is bounded by $C_{d,\varepsilon}$. Because we are looking at leaf nodes in order of increasing distance from q, looking at leaf nodes after the $C_{d,\varepsilon}$ closest leaf nodes cannot yield a point closer than one obtained from the $C_{d,\varepsilon}$ closest nodes.

Lemma 4 *Let S be the set of the $O(C_{d,\varepsilon})$ closest nodes (of any level in the quadtree) to the query point q and T be the set of the $C_{d,\varepsilon}$ closest leaf nodes to q. Then, every leaf node $t \in T$ is a descendant of or equivalent to some node $s \in S$*

Proof. Assume that this assertion is false, Then, there is at least one leaf $t_i \in T$ that is not contained any of the nodes of S. The node t_i must therefore be descended from some other node u, higher in the quadtree, such that $x \notin S$. However, t_i is one of the $C_{d,\varepsilon}$ closest leaf nodes to q. The shortest distance from q to x can be no farther than the distance from q to t_i. If we apply this to all other $t_j \in T$ and the respective $s_j \in S$ that contains t_j, then x can be at worst the $C_{d,\varepsilon}$ farthest node out of all the nodes in S, which is a contradiction because $x \notin S$ and S consists of the $O(C_{d,\varepsilon})$ closest nodes to q. Therefore, x must be in S, and every $t \in T$ is a descendent of some $s \in S$. □

Thus, we only have to maintain the $O(C_{d,\varepsilon})$ closest nodes during the algorithm because these nodes are the ancestors of the $C_{d,\varepsilon}$ closest leaf nodes, in which the approximate nearest neighbor must be contained. A pointer to the description of a node in the DFUDS representation of the quadtree only costs $O(\log m)$ bits, so the priority queue costs $O(C_{d,\varepsilon} \log m) = o(m)$ bits to store.

It takes at most $O(\log \Phi)$ time to traverse the quadtree and reach a leaf node and $O(\lg^2 \lg m)$ to access points of a node in the leftover array, and there are $C_{d,\varepsilon}$ leaves that the algorithm will access, so the running time of the algorithm is $O(C_{d,\varepsilon} \log \Phi \lg^2 \lg m)$.

5.1 Approximate k-Nearest Neighbor

We can extend the solution to the single nearest neighbor problem to solve the approximate k-nearest neighbors problem, in which we wish to return k points such that the ith point returned, p'_i, satisfies $dist(q, p'_i) \leq (1+\varepsilon) dist(q, p_i)$, where p_i is the actual ith closest point to q. As described in [3], the number of leaves that must be looked at is $O(C_{d,\varepsilon} + k)$, so the running time and space usage (in bits) of the algorithm become $O((1/\varepsilon^d + k) \log \Phi \lg^2 \lg m)$ and $O(C_{d,\varepsilon} \log m) = o(m)$, respectively. We also maintain the k-closest points to q encountered so far, and as in the previous case, these points will be approximate k-nearest neighbors after the algorithm terminates. We maintain these

points in a queue that is ordered by decreasing distance from q and occupies kb bits of space. If we find a point that is closer to q than any of the k so far, we insert it into the queue and remove the kth point.

6 Dynamic Case

We now provide a brief sketch of the techniques employed in insertion. First, we show that the DFUDS bitstring requires only $O(\log \Phi)$ time to be updated. Then, we represent the leftover array as a dynamic array, as in [15], so that we can efficiently add or remove bits from the leftover array. Finally, we implement an *update array* that allows us to update the values stored in the auxiliary data structures so that we can accurately answer queries after inserting a new point.

6.1 Insertion

In this section, we describe the process by which a new point p in the unit hypercube can be inserted into the data structure. In a standard quadtree, the general procedure for inserting p consists of traversing the quadtree until a leaf node is reached that contains p. If this leaf does not contain any other point, then we associate it with p without making any modifications. If there does exist a point inside the leaf, then we split the leaf and perform a quadtree decomposition on that leaf and the points inside it until p and the other points are separated into different leaf nodes. In the case of the succinct quadtree, must update the DFUDS bit string of the quadtree, the leftover array, and all other auxiliary data structures.

In addition to the changing information in the data structures, the size of the parts of various data strucutres are dependent on the number of points. That is, as the value of $\log n$ changes upon insertion or deletion of points, the data structures depending on this value must also structurally change. We do not deal with this issue in this paper, but instead direct the reader to [12], where a thorough solution is developed such that the sizes of the substructures can be updated without affecting the running times of any other algorithms presented. We now describe methods for updating each of the data structures to reflect the insertion of p.

6.2 DFUDS bit string

As previously mentioned, the basic idea for inserting p is to traverse the quadtree until the leaf node u containing p is reached. Because we require that every leaf node is associated with exactly one point, u already contains some point q and we must recursively split u into child nodes until p and q are in different leaves. When we split u, at most two new child nodes will be created, if p and q are in different child nodes. Inserting a new node into the tree structure requires the insertion of a pair of matching parentheses (that is, a pair of bits, 1 and 0) into the DFUDS bit string. This can be performed in $O(1)$ amortized time (see Davoodi and Rao [7]).

However, we must also consider the worst case in which p and q require many recursive splits to be separated into different leaves. The height of the tree can be at most $O(\log \Phi)$, which is the maximum number of recursive splits necessary to separate p and q. If each split adds $O(1)$ new nodes to the tree and it takes $O(1)$ time to insert a single pair of parentheses into the DFUDS bitstring, then the worst case time to update the DFUDS bit string is $O(\log \Phi)$ amortized time.

6.3 Leftover array

Upon inserting p, we must update the leftover bit array to include the leftover bits of p. If p fell into a leaf node that was occupied by a point q, then the resulting splits will increase q's level as well and therefore reduce the number of leftover bits for q. Since only two points are involved, only $O(b)$ bits of the leftover array will be changed. The running time of updating the leftover array will be $O(b) \cdot T$, where T is the time to update a single bit of the leftover array.

The problem of updating the leftover array is quite similar to the well-studied dynamic array problem. In the dynamic array problem, we are given an array and we wish to access, insert, or delete elements at any position. Raman *et al.* obtained optimal results for this problem [15], with $O\big(\frac{\lg n}{\lg \lg n}\big)$ time for insert, delete, and access operations on an array of length n, while using $o(n)$ extra bits. The length of the leftover array contains $O(bn)$ bits, so we can represent the leftover array as a dynamic array that supports the desired operations in $O\big(\frac{\lg bn}{\lg \lg bn}\big)$ time and $o(bn)$ bits. Therefore, it takes $O\big(b\frac{\lg bn}{\lg \lg bn}\big)$ time in total to update the leftover array after the insertion.

6.4 Auxiliary Data Structures

As previously mentioned, we must also update any auxiliary data structures to reflect any new points that have been inserted. The auxiliary data structures associated with the DFUDS bit string already are adapted for the dynamic case [7], so we will not discuss them further. We now develop a dynamic representation of the *leaf-sum*(i) data structure presented in Section 3.1.

The *leaf-sum*(i) structure consists of two levels of subdivision that need to be updated upon insertion of a new leaf. First, we must update the values stored at the head of each miniblock that precede the inserted node within the same block. There are $O(t/s) = O(\log m)$ miniblocks within a block. We simply iterate through each one that precedes the newly inserted node and add one to the stored value. This takes $O(\log m)$ time, but requires no additional space.

Next, we must update the values stored at the head of each block that come after the newly inserted node. There are too many blocks to iterate through, so we implement another auxiliary data structure, called the *update array* U, that contains an element corresponding to each block. When we insert a leaf node into block i, we need to add one to all the values stored at the heads of the blocks that come after block i to account for the new leaf. Instead of directly adding one to the value at the heads of the blocks, we add one to the value stored at position i in the U (likewise, we subtract one for deletion). Element $U[i]$ contains the number of additional leaf nodes present in block i that are not accounted for in the value stored at the head of the block. When we are returning a *leaf-sum*(i) value, we first locate the preceding block and obtain the value stored at its head. Then, we look at all the elements up to and including that block in U and sum the values up to the position corresponding to the block. The final returned value is the sum of the value stored at the head of the block and the accumulated sum from U. U can be dynamically maintained to support the cumulative sum operation and an update operation by representing it as a dynamic partial sums array of [7]. Under this representation, the desired operations take $O(\lg^2 \lg m)$ time and require $O\big(m \frac{\log\log m}{\log^2 m}\big) = o(m)$ bits of additional space.

6.5 Point Deletion

Point deletion is largely symmetrical to point insertion. To delete a point p from our structure, we first locate the leaf node that contains p. We remove this leaf node from the DFUDS representation of the quadtree (as shown in [7]) and follow the same path back up the tree that was traversed when locating p. We continue to delete nodes along this path up the tree until we reach a node that has a sibling. At each step, we are deleting a node, so as in the previous section, we must update the three mentioned data structures. Each of theses structures supports both insertion and deletion of their respective elements, as noted above. At each step of the traversal up the tree, we are deleting a node, which takes $O\big(b\frac{\log bn}{\log\log bn} + \log m\big)$ time. The number of nodes deleted is no more than the height of the tree, so the total time required to perform the delete operation is $O\big(\log\Phi\big(b\frac{\log bn}{\log\log bn} + \log m\big)\big)$.

7 Conclusions

In this paper, we presented a succinct variant of the quadtree that requires only $Z + o(Z)$ bits to store. Using the succinct quadtree, points can be inserted and deleted in roughly polylogarithmic time and dynamic approximate range counting and approximate nearest neighbor queries can be answered in the times specified in Theorems 1 and 2.

There is significant improvement to be made in reducing the various logarithmic factors. Furthermore, only empirical tests can confirm the theoretical improvements to the space efficiency of the quadtree in practice. Empirical results may reveal that more space efficient data structures for geometric queries will yield improvements in running time because of reduced memory usage and increased memory locality of data.

The algorithms and data structures presented in this paper depend on the aspect ratio Φ of the point set, which may result in slow running times for extreme data sets. Imposing reasonable and empirically determined restrictions on Φ may yield running times more comparable to those of algorithms that do not depend on the distribution of the points. It should also be noted that several quadtree-based algorithms traverse the tree by moving only one edge at a time. A method for compressing paths or moving over multiple edges at once using a succinct structure may speed up the many algorithms that rely on traversal of the quadtree.

8 Acknowledgments

We would like to express our gratitude to one of the anonymous reviewers, who pointed out an error in the processing of the leaf-sum queries.

References

[1] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. ALENEX'10*, pages 84–97, 2010.

[2] S. Arya and D. M. Mount. Approximate range searching. *Comput. Geom. Theory Appl.*, 17:135–163, 2001.

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

[4] D. Benoit, E. D. Demaine, J. I. Munro, and V. Raman. Representing trees of higher degree. In *Algorithms and Data Structures*, pages 169–180. Springer, 1999.

[5] P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *Proc. 16th Internat. Workshop on Algorithms and Data Structures*, volume 5664 of *Lecture Notes Comput. Sci.*, pages 98–109. Springer-Verlag, 2009.

[6] T. Chan. A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions. (Unpublished. Available from `http://www.cs.uwaterloo.ca/~tmchan/pub.html`), 2006.

[7] P. Davoodi and S. S. Rao. Succinct dynamic cardinal trees with constant time operations for small alphabet. In *Theory and Applications of Models of Computation*, pages 195–205. Springer, 2011.

[8] R. Graham, D. Knuth, and O. Patashnik. Concrete mathematics: A foundation for computer science. *Addison & Wesley*, 1989.

[9] S. Har-Peled. *Geometric Approximation Algorithms*. American Mathematical Soc., 2011.

[10] B. Hudson. Succinct representation of well-spaced point clouds. *CoRR*, abs/0909.3137, 2009.

[11] G. Jacobson. Space-efficient static trees and graphs. In *30th Annual Symp. on Foundations of Computer Science*, pages 549–554. IEEE, 1989.

[12] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. on Algorithms*, 4(3):32, 2008.

[13] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[14] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. on Computing*, 31(3):762–776, 2001.

[15] R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In *Algorithms and Data Structures*, pages 426–437. Springer, 2001.

[16] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 134–149. Society for Industrial and Applied Mathematics, 2010.

[17] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, 2006.