A Unified Approach to Approximate Proximity Searching

Sunil Arya[1,⋆], Guilherme D. da Fonseca[2,⋆⋆], and David M. Mount[3,⋆⋆⋆]

[1] Department of Computer Science,
The Hong Kong University of Science and Technology, Hong Kong
arya@cs.ust.hk
[2] Universidade Federal do Estado do Rio de Janeiro (Unirio), Brazil
fonseca@uniriotec.br
[3] Department of Computer Science and Institute for Advanced Computer Studies,
University of Maryland, College Park
mount@cs.umd.edu

Abstract. The inability to answer proximity queries efficiently for spaces of dimension $d > 2$ has led to the study of approximation to proximity problems. Several techniques have been proposed to address different approximate proximity problems. In this paper, we present a new and unified approach to proximity searching, which provides efficient solutions for several problems: spherical range queries, idempotent spherical range queries, spherical emptiness queries, and nearest neighbor queries. In contrast to previous data structures, our approach is simple and easy to analyze, providing a clear picture of how to exploit the particular characteristics of each of these problems. As applications of our approach, we provide simple and practical data structures that match the best previous results up to logarithmic factors, as well as advanced data structures that improve over the best previous results for all aforementioned proximity problems.

1 Introduction

The term *proximity* refers informally to the quality of being close to some point or object. Proximity data structures arise from numerous applications in science and engineering because it is a fundamental fact that nearby objects tend to exert a greater influence and have greater relevance than more distant objects. The inability to answer proximity queries efficiently for spaces of dimension $d > 2$ has motivated study of approximate solutions to proximity problems. In recent years, a number of different techniques have been proposed for solving these problems. In order to obtain the best performance, the technical elements

⋆ Research supported by the Research Grants Council of Hong Kong, China under project number 610106.
⋆⋆ Research supported by CNPq grant PDJ-151194/2007-6, FAPERJ grant E-26/110.091/2010 and CNPq/FAPERJ grant E-26/110.552/2010.
⋆⋆⋆ Research supported by NSF grant CCR-0635099 and ONR grant N00014-08-1-1015.

M. de Berg and U. Meyer (Eds.): ESA 2010, Part I, LNCS 6346, pp. 374–385, 2010.

of these solutions vary considerably, depending on the type of problem being solved and the properties of the underlying entities. Each variant involves its own particular construction and analysis. Consequently, it is difficult to obtain a clear understanding of the basic mechanisms underlying these approaches. In this paper, we present efficient solutions for several approximate proximity problems, all within a simple and unified framework.

Abstractly, a *proximity search problem* involves preprocessing a multidimensional point set into a data structure in order to efficiently answer queries based on distances. Consider a set P of n points in d-dimensional Euclidean space, for a constant $d \geq 2$. In problems involving aggregation, it is useful to associate each point $p \in P$ with a weight, $w(p)$, which is assumed to be drawn from some commutative semigroup $(\mathbf{S}, +)$. For example, range queries involve computing the semigroup sum of points lying within some region. In such cases, properties of the semigroup may be exploited in order to obtain the most efficient solution. An important semigroup property is *idempotence*, which means that $x + x = x$, for all $x \in \mathbf{S}$. (As an example, consider the integers under maximum or minimum.)

In this paper we consider the following fundamental proximity problems. In each case, we are given a query point q and possibly a radius r. We present each problem in its exact form, but each has a natural approximation version, given an approximation parameter $0 < \varepsilon < 1$. Points within distance $r(1-\varepsilon)$ of q must be considered and only points within distance $r(1+\varepsilon)$ may be considered.

- *Spherical range queries*: Given a query point q and a radius r, determine the number of points of P (or more generally, the semigroup sum of the weights of the points) lying within distance r from q. If the semigroup is idempotent, this is called an *idempotent spherical range query.*
- *Spherical emptiness queries*: This can be viewed as a special case of an idempotent range query in which the goal is to determine whether any point of P lies within distance r of q.
- *Nearest neighbor queries*: Given a query point q determine the closest point of P to q and its associated distance r.

Spherical emptiness queries and nearest neighbor queries are strongly related. Given the nearest neighbor p of a query point q, we can determine whether the ball of radius r centered at q is empty by comparing r and $\|pq\|$. In the approximate version, we can answer nearest neighbor queries by computing a constant-factor approximation of the nearest neighbor and then performing a binary search with $O(\log \frac{1}{\varepsilon})$ spherical emptiness queries [19]. In this paper, we will present methods for answering approximate spherical emptiness queries, and results for approximate nearest neighbor queries will follow as corollaries.

Prior results. Approximate nearest neighbor queries in spaces of fixed dimension have been widely studied. Data structures with $O(n)$ storage space and query times no better than $O(\log n + 1/\varepsilon^{d-1})$ have been proposed by several authors [8, 9, 11, 15]. In subsequent papers, it was shown that query times could be reduced, at the expense of greater space [10, 19, 13, 23]. Although different tradeoffs were achieved, in all cases the products of the ε terms in the storage and query times

are roughly $O(1/\varepsilon^{d-1})$. These space-time tradeoffs were improved in [3, 2, 6]. It was shown that $O\big(\log n + 1/\varepsilon^{\frac{d-1}{2}}\big)$ query time could be achieved with essentially $O(n)$ storage, and generally it is possible to achieve space-time tradeoffs where the product of the ε terms in the storage and the *square* of the query time is roughly $O(1/\varepsilon^{d-1})$.

Early results on approximate range searching for general semigroups provided query times of $O(\log n + 1/\varepsilon^{d-1})$ with $O(n)$ space [7]. Space-time tradeoffs for idempotent spherical range searching were presented in [5]. It was shown there that, given a tradeoff parameter $\gamma \in [1, 1/\varepsilon]$, queries can be answered in time $O\big(\log n + (\log \frac{1}{\varepsilon})/(\varepsilon\gamma)^{\frac{d-1}{2}}\big)$ with $O(n\gamma^d/\varepsilon)$ storage. The tradeoff was later extended to handle spherical range queries for arbitrary semigroups in $O(\log(n\gamma) + 1/(\varepsilon\gamma)^{d-1})$ time with $O(n\gamma^d \log \frac{1}{\varepsilon})$ storage [4]. An approach for arbitrary semigroups that is more similar in spirit to ours is presented in [1], but it is limited to query times in $\Omega\big(\log n + 1/\varepsilon^{\frac{d-1}{2}}\big)$, and it does not benefit from the assumptions of idempotence or emptiness.

The best space-time tradeoffs for answering approximate nearest neighbor queries are based on *Approximate Voronoi Diagrams* (or *AVDs*) [19, 6]. While answering an approximate nearest neighbor query with an AVD consists of a simple point location in a quadtree, building and analyzing the AVDs require relatively sophisticated machinery, including WSPDs, sampling, BBD-trees, separation properties, bisector sensitivity, and spatial amortization. To extend AVDs to handle spherical range queries, several new tools were introduced [5, 4]. The cells of the AVD are divided into three different types, with different query-answering mechanisms and analyses for each.

Our results. We present a new and unified approach to proximity searching. We place all the aforementioned proximity queries within a unified framework, providing a clear picture of how to exploit the peculiarities of each problem. We do so without using most of the AVD machinery, thus obtaining data structures that are easy both to implement and analyze. As applications of our approach, we provide simple and practical data structures that match the best previous tradeoffs up to logarithmic factors, as well as advanced data structures that improve over the best previous results for all aforementioned proximity problems.

Our approach is based on a well-known data structure, the compressed quadtree (described in Section 2.1). To perform a search involving a query ball b, the search algorithm begins by computing a constant number of cells in the compressed quadtree that cover b. Each cell locally answers the query for the portion of the ball that lies within the cell. To achieve low storage, we divide the cells into two types:

(i) Cells enclosing a large number of points store a data structure whose storage is not dependent on that number. We call this data structure an *insensitive module*. The insensitive module is generally a table where a query is answered by performing a single lookup.

(ii) Cells with a small number of points store a data structure whose storage benefits from the low number of points. We call this data structure an *adaptive module*. The adaptive module can either be as simple as a list of points

where queries are answered by brute force or as complex as the most efficient data structures known for exact range searching.

Since our framework is modular, we can plug the appropriate building blocks to obtain different data structures. By plugging simple and practical data structures, we obtain bounds that match the best known bounds up to logarithmic factors. Alternatively, we can plug advanced exact data structures in order to obtain small improvements to the most efficient data structures known for all aforementioned proximity problems. For example, with $\widetilde{O}(n)$ storage, we can perform approximate nearest neighbor queries in $\widetilde{O}\big(1/\varepsilon^{\frac{d-1}{2}}\big)$ time using only simple data structures, matching the best bound previously known [3], or $\widetilde{O}\big(1/\varepsilon^{\frac{d-3}{2}+\frac{2}{d+1}}\big)$ time by using exact data structures for halfspace emptiness queries from [22]. (Throughout, $\widetilde{O}(x)$ stand for $O(x \text{ polylog}(n, 1/\varepsilon))$.) As another example, we can answer idempotent spherical range queries in polylogarithmic time with $\widetilde{O}(n/\varepsilon^{d+1})$ storage using only simple data structures, thus matching the best bound previously known [5], or with $\widetilde{O}(n/\varepsilon^{d})$ storage using exact data structures for halfspace range searching from [21].

Next, we highlight the most efficient tradeoffs obtained using our approach. The tradeoffs are described as a function of the tradeoff parameter $\gamma \in [1, 1/\varepsilon]$. Although the improvements are not dramatic, they are significant because they show that our method, while being both simpler and more unified, also offers new insights into the computational complexities of these problems.

- For general spherical range searching with query time $\widetilde{O}(1/(\varepsilon\gamma)^{d-1})$, we improve the best previous storage [4, 1] from $\widetilde{O}(n\gamma^d)$ to $\widetilde{O}(n\gamma^{d-1}(1+\varepsilon\gamma^2))$. This improves the storage by a factor of $\widetilde{O}(\gamma/(1+\varepsilon\gamma^2))$ for the same query time.
- For idempotent spherical range searching with query time $\widetilde{O}\big(1/(\varepsilon\gamma)^{\frac{d-1}{2}}\big)$, we improve the best previous storage [5] from $\widetilde{O}(n\gamma^d/\varepsilon)$ to $\widetilde{O}\big(n\gamma^{d-\frac{1}{2}}/\sqrt{\varepsilon}\big)$. This improves the storage by a factor of $\widetilde{O}(\sqrt{\gamma/\varepsilon})$ for the same query time.
- For nearest neighbor searching, the best previous result [3, 2, 6] has query time $\widetilde{O}\big(1/(\varepsilon\gamma)^{\frac{d-1}{2}}\big)$ with storage $\widetilde{O}(n\gamma^{d-1})$. We improve this result to query time $\widetilde{O}\big(1/(\varepsilon\gamma)^{\frac{d-3}{2}+\frac{1}{d}}\big)$ with storage $\widetilde{O}(n\gamma^{d-2})$, for even $d \geq 4$. For odd $d \geq 3$, we obtain query time $\widetilde{O}\big(1/(\varepsilon\gamma)^{\frac{d-3}{2}+\frac{2}{d+1}}\big)$ with storage $\widetilde{O}\big(n\gamma^{d-2+\frac{2}{d+1}}\big)$.

The insensitive modules are of special interest because they work in the absolute error model [14], providing more efficient tradeoffs for approximate spherical range searching in this model. The insensitive general module can be used to reduce the ε-dependency in algorithms for approximating the smallest k-enclosing disk [20] and the unit disk enclosing the most points [17].

2 Framework

In this section, we present our new approach as a general framework and show how to analyze its complexity. First, we review preliminary results on compressed

quadtrees. Second, as concrete illustration, we present a simple data structure for approximate spherical emptiness. Finally, we generalize this data structure to an abstract framework for approximate proximity searching.

2.1 Quadtrees

A *quadtree* is a hierarchical decomposition of the data points induced by a hierarchical partition of the space into d-dimensional hypercubes. The root of the quadtree corresponds to the whole set of data points. An internal node has 2^d children corresponding to the sets of points in the disjoint subdivisions of the parent hypercube. A leaf is a node which contains a single data point. A *quadtree box* is defined recursively as the original bounding hypercube or the hypercubes obtained by evenly dividing a quadtree box.

A *compressed quadtree*, is obtained by replacing all maximal chains of nodes that have a single non-empty child by a single node associated with the coordinates of the smallest quadtree box containing the data points. The size of a compressed quadtree is $O(n)$ and there are many different ways to build a compressed quadtree with n points in $O(n \log n)$ time [8, 18, 16]. Even though the height of the tree can be as much as $\Theta(n)$, we can efficiently search a compressed quadtree by using an auxiliary structure which can be a simple hierarchy of separators [18], a skip-quadtree [16], or a BBD-tree [8]. An important type of query that these auxiliary structures answer in $O(\log n)$ time is called a cell query [18]. Let T be a compressed quadtree for the set of points P. Given a query quadtree box Q, a *cell query* consists of finding the unique cell Q' in T such that $P \cap Q = P \cap Q'$, if it exists. The quadtree box Q' exists if $P \cap Q \neq \emptyset$ and Q' is unique because T is compressed.

Let v be a vertex in a compressed quadtree associated with a quadtree box $\square_v$ of diameter δ_v. Consider a grid with cells of diameter $\varepsilon\delta_v$ subdividing $\square_v$. Let c_v denote the number of non-empty grid cells (that is, those containing a point of P). Since there are $O(n)$ nodes in T and $c_v \leq (\frac{1}{\varepsilon})^d$ by a packing argument, it follows that $\sum_{v \in T} c_v = O\big(n(\frac{1}{\varepsilon})^d\big)$. The following technical lemma, which will be useful in analyzing the storage requirements of our data structures, shows that the sum is significantly smaller.

Lemma 1. *For any compressed quadtree T with n points, $\sum_v c_v = O(n \log 1/\varepsilon)$. Consequently, the number of nodes v with $c_v > \alpha$ is $O(n(\log 1/\varepsilon)/\alpha)$.*

Proof. The proof proceeds by a charging argument, where each of the $O(n)$ quadtree nodes receives up to $O(\log \frac{1}{\varepsilon})$ charges from the ancestors of the node. Assume without loss of generality that ε is a power of $1/2$ and consider an internal node v. Let S_1 be the set of quadtree nodes corresponding, by cell queries, to the non-empty grid cells of diameter $\varepsilon\delta_v$ subdividing $\square_v$. A node in S_1 may have arbitrarily small diameter because of compression, but the parent of a node in S_1 has diameter at least $\varepsilon\delta_v$. Let S_2 be the set of parents of the nodes in S_1. We have $|S_2| \geq |S_1|/2^d = c_v/2^d$. Node v assigns $c_v/|S_2| = O(1)$ charges to each cell in S_2, so the sum of charges over all nodes is equal to $\sum_v c_v$. Since a node v only receives charges from ancestors of diameter at most δ_v/ε, each node receives at most $O(\log \frac{1}{\varepsilon})$ charges. □

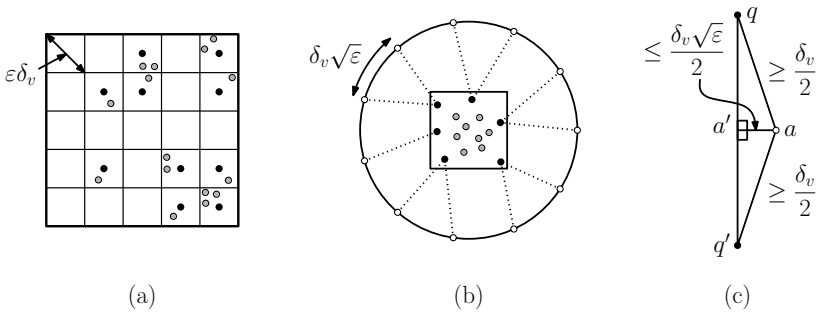

Fig. 1. Spherical emptiness cells: (a) for the case $c_v \leq 1/\varepsilon^{(d-1)/2}$ and (b) for the case $c_v > 1/\varepsilon^{(d-1)/2}$. Part (c) illustrates the correctness proof when $c_v > 1/\varepsilon^{(d-1)/2}$.

2.2 A Simple Data Structure for Approximate Spherical Emptiness

To illustrate our new approach, we demonstrate how to use it to answer approximate spherical emptiness queries. Given a query ball b of radius r and $\varepsilon > 0$, the algorithm must return "yes" if the expanded ball of radius $r(1+\varepsilon)$ is empty and "no" if the contracted ball of radius $r/(1+\varepsilon)$ is nonempty. The basis of our data structure is a compressed quadtree, together with any data structure that can answer cell queries in time $O(\log n)$. Each leaf node in the compressed quadtree simply stores the single point (if any) contained in it. Each internal node v will store a set S_v of $O(\min(c_v, 1/\varepsilon^{(d-1)/2}))$ points. We shall see that the approximate emptiness of a ball b of radius $r \geq 2\delta_v$ with respect to the points of $P \cap \square_v$ can be reduced to a simple brute-force test of whether $S_v \cap b = \emptyset$. We consider two cases, depending on the size of c_v.

If $c_v \leq 1/\varepsilon^{(d-1)/2}$, consider a grid with cells of diameter $\varepsilon\delta_v$ subdividing $\square_v$, and set S_v to be the set of at most c_v center points of the nonempty grid cells (see Figure 1(a)). Since the set S_v is obtained by moving the points in $\square_v$ by at most $\varepsilon\delta_v$ and the query ball has radius $r \geq 2\delta_v$, testing the approximate emptiness of b with respect to $P \cap \square_v$ reduces to testing whether $S_v \cap b$ is empty.

On the other hand, if $c_v > 1/\varepsilon^{(d-1)/2}$, we use a coreset construction like the one in [12]. Consider ball of radius δ_v centered at $\square_v$'s center, and let A be a β-dense set of points on the boundary of this ball, for $\beta = \delta_v\sqrt{\varepsilon}/2$. (That is, given any point a' on the boundary of the ball, there is a point $a \in A$ such that $\|aa'\| \leq \delta_v\sqrt{\varepsilon}/2$.) By a simple packing argument, we may assume that $|A| = O(1/\varepsilon^{(d-1)/2})$. For each $a \in A$, we compute its nearest neighbor among $P \cap \square_v$. Let S_v be the resulting set of nearest neighbors (see Figure 1(b)). Next, we show that this set is sufficient to approximately answer the query.

Since the case when the query ball completely contains $\square_v$ is trivial, we may assume that the center of the query ball (of radius $r \geq 2\delta_v$) is at distance at least δ_v from any point in $\square_v$. For a given query point q and its nearest neighbor q', there exists $a \in A$ such that $\|qa\| + \|aq'\| \leq \|qq'\|(1+\varepsilon)$. This follows by a simple application of the Pythagorean Theorem on the pair of right triangles defined by the obtuse triangle $\triangle qaq'$ (see Figure 1(c)). Letting

q_a denote the nearest neighbor of a among the points inside $\square_v$, it follows that $\|qq_a\| \leq \|qa\| + \|aq_a\| \leq \|qa\| + \|aq'\| \leq (1+\varepsilon)\|qq'\|$.

To answer a query for a ball b of radius r, we start by locating a set V of $O(1)$ quadtree boxes of diameter at most $r/2$ that disjointly cover $b \cap P$. This task can be performed in $O(\log n)$ time by determining the intersection of b with a grid (of side length $2^{\lfloor \log_2(r/2\sqrt{d}) \rfloor}$) and then performing the corresponding cell queries. As mentioned above, we answer the query by a simple brute-force test that, for each $v \in V$, $S_v \cap b = \emptyset$. Since we consider only a constant number of cells, each storing $O(1/\varepsilon^{(d-1)/2})$ points, the total query time is $O(\log n + 1/\varepsilon^{(d-1)/2})$.

To analyze the storage, note that, by Lemma 1 the number of nodes with $c_v > 1/\varepsilon^{(d-1)/2}$ is $O(n\varepsilon^{(d-1)/2} \log \frac{1}{\varepsilon})$. Since the storage for each node is $O(1/\varepsilon^{(d-1)/2})$, the total storage for the nodes with $c_v > 1/\varepsilon^{(d-1)/2}$ is $O(n \log \frac{1}{\varepsilon})$. The storage for each node with $c_v \leq 1/\varepsilon^{(d-1)/2}$ is $O(c_v)$. By Lemma 1, the sum of c_v for all nodes in the quadtree is $O(n \log \frac{1}{\varepsilon})$, hence the total storage is $O(n \log \frac{1}{\varepsilon})$.

Lemma 2. *Given a set of n points in d-dimensional space, there exists a data structure of space $O(n \log \frac{1}{\varepsilon})$ that can answer approximate spherical emptiness queries in time $O(\log n + 1/\varepsilon^{(d-1)/2})$, and approximate nearest neighbor queries in time $O(\log n + (\log \frac{1}{\varepsilon})/\varepsilon^{(d-1)/2})$.*

2.3 Abstract Framework

We now introduce our framework in an abstract setting. The basis of the data structure is a compressed quadtree equipped with an auxiliary structure to answer cell queries. Each internal node v in the compressed quadtree stores a data structure (called a module) that answers approximate spherical queries for a ball of radius $r \geq 2\delta_v$ with respect to the points that are inside $\square_v$. If v is a leaf node, then we simply store the single point contained in $\square_v$.

Define a *module* to be a data structure that takes two inputs: a box $\square$ and a set of points X contained in this box. A module answers ε-approximate spherical queries with respect to X for balls of radius 2δ, where δ is the size of $\square$. Let $k = |X|$, and let $s(k)$ denote the module's storage bound. We distinguish between two types of modules. A module is *insensitive* if its storage $s(k)$ depends only on ε, regardless of the number of points in $\square$. For the remaining modules, called *adaptive*, $f(k) = s(k)/k$ is assumed to be a nondecreasing function of k. For a threshold parameter α to be specified, if $c_v > \alpha$, then node v stores an insensitive module for the box $\square_v$ and the set of points $P \cap \square_v$. Otherwise, v stores an adaptive module for the box $\square_v$ and the set of center points of the grid cells defining c_v.

To answer a query for a ball of radius r, we start by identifying a set V of $O(1)$ quadtree boxes of diameter at most $r/2$ that disjointly cover the points in the query ball, in $O(\log n)$ time. Then, we answer the query inside each of these cells and combine the results. For the sake of approximate nearest neighbor queries, it is useful to observe that, if we are performing multiple queries with consecutive spheres whose radii vary within a constant factor of each other, this $O(\log n)$ term is incurred only for the first query.

Let t denote the query time for the module with the largest query time used. Since we are performing a constant number of queries among the quadtree cells after $O(\log n)$ time to perform cell queries, our data structure has query time $O(\log n + t)$. We generally choose the modules in a manner to make the query time equal to t for all modules, unless this is not an option in the tradeoff.

Let S denote the storage for each node that uses the insensitive module. Note that this is the same for all nodes, regardless of the number of points in the node. Since the insensitive module is used only for nodes v with $c_v > \alpha$, by Lemma 1, the total storage for the insensitive-module nodes is $O\big(\frac{nS}{\alpha} \log \frac{1}{\varepsilon}\big)$.

Let $s(c_v)$ denote the storage for the adaptive module with query time t storing the c_v points in $\square_v$. Recall that $c_v \leq \alpha$ for all nodes v where the adaptive module is used. Since $f(c_v) = s(c_v)/c_v$ is monotonically increasing in c_v for an adaptive module, we have $s(c_v)/c_v \leq s(\alpha)/\alpha$. Thus, the total storage for all the adaptive module nodes is $\sum_{v \in T,\, c_v \leq \alpha} s(c_v) = \sum_{v \in T} \frac{s(c_v)}{c_v} \cdot c_v \leq \frac{s(\alpha)}{\alpha} \sum_{v \in T} c_v$, which by Lemma 1 is $O\big(\frac{ns(\alpha)}{\alpha} \log \frac{1}{\varepsilon}\big)$. In summary, we have:

Theorem 1. *For any threshold parameter α, given an insensitive module with query time t and storage S, and an adaptive module with query time t and storage $s(\alpha)$ for α points, we can build a spherical range searching data structure (for a semigroup compatible with the modules) with $O(\log n + t)$ query time and storage*

$$O\left(\frac{n(S + s(\alpha))}{\alpha} \log \frac{1}{\varepsilon}\right).$$

3 Modules

In this section, we consider the design of efficient modules to be used in our framework. Recall that a module is a data structure for the following simple range searching problem: Preprocess a set of n points, inside a box of diameter δ, in order to efficiently answer approximate spherical queries where the radius of the query ball is at least 2δ. We remark that, when addressing the design of modules, n refers to the number of points stored just in the module.

We consider three cases, general spherical range queries, idempotent queries and emptiness queries. We present each of these cases in the next three sections.

3.1 General Spherical Range Queries

In the most general version, we cannot assume any properties about the commutative semigroup. The modules designed for the general version can also be used for the idempotent or emptiness versions. The simplest (and surprisingly useful) adaptive module has both $O(n)$ storage and query time. We call this a *brute force module*, and it consists of a list of the n points where queries are answered in $O(n)$ time by inspecting each point individually.

A much more sophisticated adaptive module consists of reducing spherical range searching to halfspace range searching by lifting the points onto a $(d+1)$-dimensional paraboloid and then using Matoušek's exact range searching data

structure [21]. Storage is $m \in [n, n^{d+1}/\log^{d+1} n]$, preprocessing is $O(n^{1+\beta} + m\log^{\beta} n)$ for arbitrarily small β, and query time is $O(n/m^{1/(d+1)})$. We call this an *exact general module*. Next, we describe an insensitive module.

Let $x_1, \ldots, x_d$ denote the orthogonal axes. We call the x_d axis *vertical*, the hyperplane determined by the remaining axes *horizontal*, and use standard terms such as *top* with respect to these directions. Without loss of generality, we assume that the box of diameter δ intersects only a portion of the query ball boundary that has normal vectors within an angle at most $\pi/4$ of the vertical axis. Separate data structures can be defined for a constant number of rotated sets of points.

Consider a $d-1$ dimensional grid subdividing the horizontal hyperplane into cells of diameter $\rho \leq \delta$, for a parameter ρ to be defined later. Each of these cells induces a prism inside the bounding hypercube in the vertical direction, which we call a *column*. The column has height $O(\delta)$ and size $O(\rho)$ in the horizontal directions. Next, we define a data structure to answer approximate spherical range queries, in time $O(1)$, corresponding to the portion of the query ball that is contained in a column. The number of columns in the box, which we will denote by C, is $O((\delta/\rho)^{d-1})$.

For simplicity, we scale down the space by ρ, making the horizontal diameter of the column equal to 1 and the height $\delta' = \Theta(\delta/\rho)$. We describe a data structure with absolute approximation error ϕ, irrespective of the radius of the query ball (as in the absolute error model [14]).

Let B be a ball of radius $r \geq 2$ centered on the vertical axis and tangent to the horizontal hyperplane $x_d = 0$. We define $f(r) = r - \sqrt{r^2 - 1}$ as the length of a vertical segment connecting B to a point at distance 1 from the origin in the horizontal hyperplane $x_d = 0$. Consider two balls of radius r and r' that are tangent to each other at a point within the column. If $|f(r') - f(r)| \leq \phi$, then it is easy to see that these balls approximate each other inside the column, in the sense that any point on intersection of the column with the boundary of one ball is within (vertical) distance of ϕ of a point on the boundary of the other ball. Let R be the set of radii $r \leq 1/\phi$ such that $f(r)$ is a multiple of ϕ.

Next, we analyze $|R|$. It is easy to show that $f(r) \leq 1/r$ (by setting $r = \sec x$ with $0 \leq x < \pi/2$, and applying standard trigonometric identities). Since $r \geq 2\delta'$, we have $f(r) \leq 1/\delta'$, and therefore $|R| = O(1 + 1/(\delta'\phi))$.

Let G be a set of balls of radius r', for each $r' \in R$, centers with vertical coordinates in increments of ϕ, and centers with horizontal coordinates defined by the vertices of a horizontal grid with cells of diameter $\phi r'$. Consider that G only has balls such that the normal vectors inside the column are within an angle at most $\pi/4$ of the vertical axis, since other balls can be handled using a suitable rotation. We always have a ball $g \in G$ whose boundary is within vertical distance ϕ of a tangent ball to the query ball inside the column. The set of balls G define a data structure with $O(1)$ query time for query balls of radius $r \geq 2\delta'$ inside a column of horizontal diameter 1. For each radius r', the set G has $H = O(1/\phi^{d-1})$ horizontal coordinates and, since we only store balls that have normal vectors within an angle at most $\pi/4$ of the vertical axis, G has $V = O(\delta'/\phi)$ vertical coordinates, $|G| = |R|HV = O\big(\big(1 + \frac{1}{\delta'\phi}\big)\frac{\delta'}{\phi^d}\big)$.

To change the scale back to columns of diameter ρ with error $\varepsilon\delta$, we set $\delta' = \delta/\rho$ and $\phi = \varepsilon\delta/\rho$, from which we have $|G| = O\big(\big(1 + \frac{\rho^2}{\varepsilon\delta^2}\big)\frac{\rho^{d-1}}{\varepsilon^d\delta^{d-1}}\big)$.

To obtain the data structure for the whole bounding box, we build a set of balls for each column, cropping the balls to inside the corresponding columns. Let $\gamma \in [1, 1/\varepsilon]$ be a parameter to control the space-time tradeoff. Setting $\rho = \delta\varepsilon\gamma$, and multiplying by the $C = O((\delta/\rho)^{d-1})$ columns we have storage $S = C|G| = O\big(\big(1 + \varepsilon\gamma^2\big)\frac{1}{\varepsilon^d}\big)$.

Next, we describe an adaptive module based on the previous idea. Instead of having $V = O(1/\varepsilon)$ balls in $C = O(1/(\varepsilon\gamma)^{d-1})$ columns, we only consider balls that have a point on the boundary of the ball. Therefore, if the box has n points, the total storage is $n|R|H = O(n(1 + \varepsilon\gamma^2)\gamma^{d-1})$. The query time increases by an $O(\log \frac{1}{\varepsilon})$ factor because we need to perform a binary search for each column.

Lemma 3. *There are adaptive modules for the general version with (i) query time and storage $t = s(n) = O(n)$ (brute force module), (ii) query time $t = O(n/m^{1/(d+1)})$ and storage $m \in [n, n^{d+1}/\log^{d+1} n]$ (exact general module), and (iii) query time $t = O((\log \frac{1}{\varepsilon})/(\varepsilon\gamma)^{d-1})$ and storage $S = O(n(1 + \varepsilon\gamma^2)\gamma^{d-1})$, for $\gamma \in [1, 1/\varepsilon]$ (approximate general module). There is an insensitive module for the general version with query time $t = O(1/(\varepsilon\gamma)^{d-1})$ and storage $S = O((1 + \varepsilon\gamma^2)/\varepsilon^d)$, for $\gamma \in [1, 1/\varepsilon]$ (insensitive general module).*

Applications. The modules from Lemma 3 can be used together with Theorem 1 to obtain the following data structures for spherical range searching. If we use the insensitive general module and the brute force module, setting $\alpha = 1/(\varepsilon\gamma)^{d-1}$, then we obtain query time $O(\log n + \alpha)$ with storage $O(n\gamma^{d-1}(1+\varepsilon\gamma^2)(\log \frac{1}{\varepsilon})/\varepsilon)$. If we use only the approximate general module (by setting $\alpha = n$), then we have query time $O(\log n + (\log \frac{1}{\varepsilon})/(\varepsilon\gamma)^{d-1})$ with storage $O(n\gamma^{d-1}(1 + \varepsilon\gamma^2)\log \frac{1}{\varepsilon})$.

3.2 Idempotent Spherical Range Queries

It is not known how to exploit idempotence in exact range searching. Thus, we introduce no adaptive module for this case. The insensitive module for the idempotent case is strongly based on the insensitive general module, albeit much more efficient. In the idempotent version, the generators can overlap, therefore we do not need to crop the balls inside each column as in the insensitive module. A careful look at the insensitive module shows that the same balls are used by essentially all columns. Therefore, the storage for a single column is equal to the total storage. In the idempotent and emptiness versions, we set $\rho = \delta\sqrt{\varepsilon\gamma}$ to obtain query time $O(1/(\varepsilon\gamma)^{(d-1)/2})$. We have storage $S = |G| = O((\gamma/\varepsilon)^{(d+1)/2})$.

Lemma 4. *There is an insensitive module for the idempotent version with query time $t = O(1/(\varepsilon\gamma)^{(d-1)/2})$ and storage $S = O((\gamma/\varepsilon)^{(d+1)/2})$, for $\gamma \in [1, 1/\varepsilon]$ (insensitive idempotent module).*

Applications. The insensitive idempotent module from Lemma 4 can be used together with Theorem 1 to obtain the following data structures for the idempotent

version. If we use brute force as the adaptive module and set $\alpha = 1/(\varepsilon\gamma)^{(d-1)/2}$, then we obtain query time $O(\log n + \alpha)$ with storage $O(n\gamma^d(\log \frac{1}{\varepsilon})/\varepsilon)$. If we use the exact general module and set $\alpha = 1/\varepsilon^{\frac{d}{2}}\gamma^{\frac{d-2}{2}}$, then we obtain query time $O(\log n + 1/(\varepsilon\gamma)^{\frac{d-1}{2}})$ with storage $O(n\gamma^{d-\frac{1}{2}}(\log \frac{1}{\varepsilon})/\sqrt{\varepsilon})$.

3.3 Spherical Emptiness Queries

Halfspace emptiness is a well studied problem. Exact data structures for halfspace emptiness are much more efficient than for general semigroups. By lifting the points onto a $(d+1)$-dimensional paraboloid and then using Matoušek's halfspace emptiness data structure [22] we obtain an adaptive module with storage $m \in [n, n^{\lceil d/2 \rceil}]$ and query time $\widetilde{O}(n/m^{1/\lceil d/2 \rceil})$. We call this an *exact emptiness module.*

To obtain an insensitive module for the emptiness version, we simply modify the insensitive idempotent module in order to store only the vertical coordinate of the center of the bottommost non-empty ball for each horizontal coordinate of the center. Therefore, storage is reduced by a factor of $O(1/\varepsilon)$. Storage becomes $O(\gamma^{(d+1)/2}/\varepsilon^{(d-1)/2})$ with the same $O(1/(\varepsilon\gamma)^{(d-1)/2})$ query time. Note that this insensitive module generalizes the one used in Section 2.2.

We can improve upon this by adapting constructs from [6] (such as the Concentric Ball Lemma) and using exact data structure for spherical emptiness. As a result, it is possible to obtain an insensitive module whose storage is $O(\gamma^{(d+1)/2}/\varepsilon^{(d-1)/2})$ and whose query time is $\widetilde{O}(1/(\varepsilon\gamma)^{(d-3)/2+1/d})$ for even d and $\widetilde{O}(1/(\varepsilon\gamma)^{(d-3)/2+2/(d+1)})$ for odd d. The technical details are omitted.

Lemma 5. *There is an adaptive module for the emptiness version with query time $t = \widetilde{O}(n/m^{1/\lceil d/2 \rceil})$ and storage $m \in [n, n^{\lceil d/2 \rceil}]$ (exact emptiness module).*

There is an insensitive module for the emptiness version with query time $t = O(1/(\varepsilon\gamma)^{(d-1)/2})$ and storage $S = O(\gamma^{(d+1)/2}/\varepsilon^{(d-1)/2})$, for $\gamma \in [1, 1/\varepsilon]$ (insensitive emptiness module). There is also an insensitive module for the emptiness version with query time $t = \widetilde{O}(1/(\varepsilon\gamma)^{(d-3)/2+1/d})$ for even $d \geq 4$ and $t = \widetilde{O}(1/(\varepsilon\gamma)^{(d-3)/2+2/(d+1)})$ for odd $d \geq 3$ and storage $S = \widetilde{O}(\gamma^{(d+1)/2}/\varepsilon^{(d-1)/2})$ (advanced insensitive emptiness module).

Applications. The modules from Lemma 5 can be used together with Theorem 1 to obtain the following data structures. If we use the brute force module, the insensitive emptiness module, and set $\alpha = 1/(\varepsilon\gamma)^{(d-1)/2}$, we have query time $O(\log n + \alpha)$ with storage $O(n\gamma^d \log \frac{1}{\varepsilon})$. If we use the exact emptiness module and the advanced insensitive emptiness module, then we obtain query time $\widetilde{O}(1/(\varepsilon\gamma)^{(d-3)/2+1/d})$ with storage $\widetilde{O}(n\gamma^{d-2})$ for even $d \geq 4$. For odd $d \geq 3$, we obtain query time $\widetilde{O}(1/(\varepsilon\gamma)^{(d-3)/2+2/(d+1)})$ with storage $\widetilde{O}(n\gamma^{d-2+2/(d+1)})$.

References

1. Arya, S., da Fonseca, G.D., Mount, D.M.: Tradeoffs in approximate range searching made simpler. In: Proc. 21st SIBGRAPI, pp. 237–244 (2008)
2. Arya, S., Malamatos, T.: Linear-size approximate Voronoi diagrams. In: Proc. 13th Ann. ACM-SIAM Symp. Discrete Algorithms (SODA), pp. 147–155 (2002)

3. Arya, S., Malamatos, T., Mount, D.M.: Space-efficient approximate Voronoi diagrams. In: Proc. 34th Ann. ACM Symp. Theory of Comput. (STOC), pp. 721–730 (2002)
4. Arya, S., Malamatos, T., Mount, D.M.: Space-time tradeoffs for approximate spherical range counting. In: Proc. 16th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 535–544 (2005)
5. Arya, S., Malamatos, T., Mount, D.M.: On the importance of idempotence. In: Proc. 38th ACM Symp. on Theory of Comput. (STOC), pp. 564–573 (2006)
6. Arya, S., Malamatos, T., Mount, D.M.: Space-time tradeoffs for approximate nearest neighbor searching. J. ACM 57, 1–54 (2009)
7. Arya, S., Mount, D.M.: Approximate range searching. Comput. Geom. 17, 135–163 (2001)
8. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. J. ACM 45(6), 891–923 (1998)
9. Bespamyatnikh, S.N.: Dynamic algorithms for approximate neighbor searching. In: Proc. 8th Canad. Conf. Comput. Geom. (CCCG), pp. 252–257 (1996)
10. Chan, T.M.: Approximate nearest neighbor queries revisited. Discrete Comput. Geom. 20, 359–373 (1998)
11. Chan, T.M.: Closest-point problems simplified on the ram. In: Proc. 13th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA), pp. 472–473 (2002)
12. Chan, T.M.: Faster core-set constructions and data-stream algorithms in fixed dimensions. Comput. Geom. 35(1), 20–35 (2006)
13. Clarkson, K.L.: An algorithm for approximate closest-point queries. In: Proc. 10th Annu. ACM Symp. Comput. Geom. (SoCG), pp. 160–164 (1994)
14. da Fonseca, G.D., Mount, D.M.: Approximate range searching: The absolute model. Comput. Geom. 43(4), 434–444 (2010)
15. Duncan, C.A., Goodrich, M.T., Kobourov, S.: Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. J. Algorithms 38, 303–333 (2001)
16. Eppstein, D., Goodrich, M.T., Sun, J.Z.: The skip quadtree: a simple dynamic data structure for multidimensional data. In: Proc. 21st ACM Symp. Comput. Geom. (SoCG), pp. 296–305 (2005)
17. Funke, S., Malamatos, T., Ray, R.: Finding planar regions in a terrain: in practice and with a guarantee. Internat. J. Comput. Geom. Appl. 15(4), 379–401 (2005)
18. Har-Peled, S.: Notes on geometric approximation algorithms, `http://valis.cs.uiuc.edu/~sariel/teach/notes/aprx/`
19. Har-Peled, S.: A replacement for Voronoi diagrams of near linear size. In: Proc. 42nd Ann. Symp. Foundations of Computer Science (FOCS), pp. 94–103 (2001)
20. Har-Peled, S., Mazumdar, S.: Fast algorithms for comput. the smallest k-enclosing circle. Algorithmica 41(3), 147–157 (2005)
21. Matoušek, J.: Range searching with efficient hierarchical cutting. Discrete Comput. Geom. 10, 157–182 (1993)
22. Matoušek, J., Schwarzkopf, O.: On ray shooting in convex polytopes. Discrete Comput. Geom. 10, 215–232 (1993)
23. Sabharwal, Y., Sen, S., Sharma, N.: Nearest neighbors search using point location in balls with applications to approximate Voronoi decompositions. J. Comput. Sys. Sci. 72, 955–977 (2006)