

Approximate Range Searching In The Absolute Error Model

by

Guilherme Dias da Fonseca

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Advisory Committee:

Professor David M. Mount, Chair
Professor Michael Boyle
Professor David W. Jacobs
Professor Samir Khuller
Professor Clyde P. Kruskal

Contents

1	Introduction	1
1.1	Summary of Results	7
1.2	Organization	13
2	Preliminaries	15
2.1	The ASA Model	15
2.2	Partition Trees and Quadtrees	18
2.3	Some Geometric Lemmas	23
3	Previous Work	29
3.1	Orthogonal Range Searching	30
3.1.1	kd-Trees	30
3.1.2	Range Trees	32
3.2	Simplex and Halfspace Range Searching with Linear Space . .	35
3.2.1	Ham Sandwich Cut	35
3.2.2	Simplicial Partitions	37
3.3	Simplex and Halfspace Range Searching in Polylogarithmic Time	38
3.3.1	Geometric Duality	39
3.3.2	VC-Dimension, ε -Nets, and ε -Approximations	40
3.3.3	Cuttings	42
3.4	Relative Error Model	44
4	Orthogonal and Convex Ranges	48
4.1	Orthogonal Range Searching	49
4.1.1	Group Version	51
4.1.2	Semigroup Version	52
4.1.3	Reporting Version	55
4.2	Convex Range Searching	56
4.3	Range Sketching	59

5	Halfspace Range Searching	62
5.1	Previous Results	63
5.2	Approximate Semigroup Version	66
5.3	Approximate Idempotent Version	70
5.4	Approximate Emptiness Version	74
5.5	Exact Idempotent Version	78
6	Halfbox Quadtree	81
6.1	Definition and Preprocessing	82
6.2	Spherical Range Searching	83
6.3	Approximate Nearest Neighbor	86
6.4	Simplex Range Searching	89
6.5	Approximate Range Reporting	93
6.6	The Data Stream Model	95
6.7	The Relative Error Model	101
7	Conclusion	109
7.1	Absolute Model Data Structures	109
7.2	Applications	111
7.3	Future Work	112
	Bibliography	115
	Index	120

List of Tables

1.1	Complexities of several approximate range searching data structures.	9
3.1	Complexities of exact orthogonal range searching data structures.	35
6.1	Query time of the reporting halfbox quadtree for different range shapes.	95
6.2	Query time of the stream halfbox quadtree for different range shapes.	101
6.3	Complexities of the relative halfbox quadtree for different range shapes.	103

List of Figures

1.1	Examples of simplex and halfspace range searching.	2
1.2	Subset relation between different range spaces.	4
1.3	Fuzzy boundaries in the absolute error model.	5
2.1	Representation of a quadtree.	19
2.2	Representation of a compressed quadtree.	20
2.3	Triangle used in the proof of Lemma 2.2.	23
3.1	Representation of a 2-dimensional kd-tree.	31
3.2	Representation of a 1-dimensional range tree.	33
3.3	Representation of a 2-dimensional range tree.	34
3.4	Set of 16 points divided into four subsets of the same size by 2 lines.	36
3.5	Example of a simplicial partition in the plane.	38
3.6	Example of primal and dual spaces.	40
3.7	Example of a BBD-tree.	45
4.1	Examples of orthogonal ranges.	49
4.2	Point approximation.	50
4.3	Array built for a set of points with weight 1 in the plane.	51
4.4	Subdividing an array recursively in a fixed dimension and linking to a $(d - 1)$ -dimensional data structure.	53
4.5	Data structure with $O(1)$ query time for the semigroup version of partial sum.	54
4.6	Examples of convex ranges.	56
4.7	Convex range query.	57
4.8	Example of range sketching.	60
5.1	Boundaries of the halfspaces in \mathcal{G} with $\varepsilon' = 0.1$	67
5.2	Idempotent version of the halfspace range searching data structure.	71
5.3	Proof of Lemma 5.3.	72

5.4	Proof of Lemma 5.6.	76
6.1	Representation of a halfbox quadtree.	82
6.2	Approximating a ball using quadtree boxes and half quadtree boxes.	84
6.3	Approximate nearest neighbor of a point q	87
6.4	Representation of a skinny triangle.	92
6.5	Fuzzy boundaries of different query ranges in the relative model.	102
6.6	Smooth range in the proof of Lemma 6.11.	105

Chapter 1

Introduction

The *range searching problem* involves preprocessing a set P of n points in \mathbb{R}^d so that given a region R , a predefined function $f(P \cap R)$ can be computed efficiently. The region R is called a *range*, and it is drawn from a predefined *range space* \mathcal{R} . We let $q(R) = f(P \cap R)$ denote the result of the *query*. The points $p \in P$ are called *data points*. Two examples of range searching are represented in Figure 1.1

Range searching is a well-studied problem in computational geometry. Excellent surveys have been written by Matoušek [43] and Agarwal and Erickson [2]. The most common examples of functions include *range counting*, where $f(S) = |S|$ and *range reporting*, where $f(S) = S$.

More generally, we have a commutative semigroup $(\mathbf{S}, +)$, and each point $p \in P$ is associated with a weight $w(p) \in \mathbf{S}$. The answer to a query is the sum of the weights of the points in $P \cap R$. This version is called the *semigroup version* of the problem. With exception of range reporting, we assume that the semigroup elements can be stored in $O(1)$ space and the sum of two semigroup

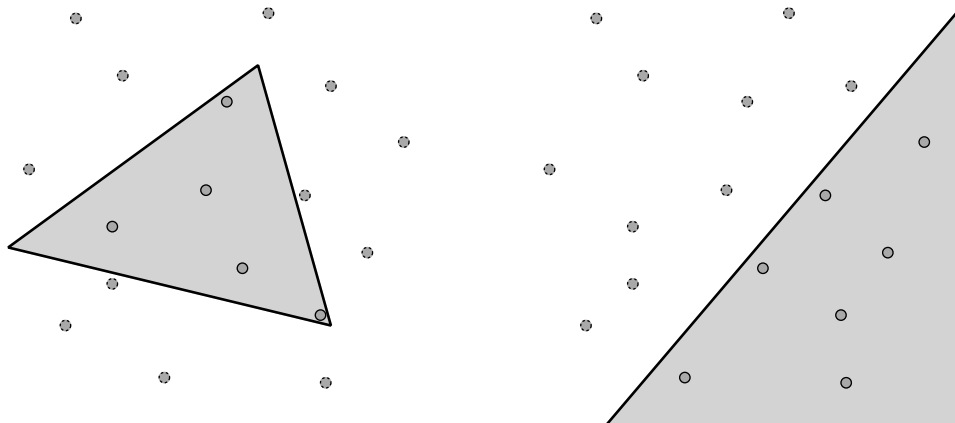


Figure 1.1: Examples of simplex range searching (left) and halfspace range searching (right). The data points inside the query ranges are solid and the data points outside the query ranges are dotted.

elements can be computed in $O(1)$ time.

In the *group version* of the problem, we assume that $(\mathbf{S}, +)$ is not only a commutative semigroup, but also a commutative (Abelian) group. The group version often admits more efficient solutions, because the presence of inverses means that both addition and subtraction may be used to compute the answer to the query. The range counting problem can be formulated in the group version, using the group $(\mathbb{Z}, +)$ where $+$ is the standard arithmetic addition, and setting $w(p) = 1$ for all $p \in P$. Note that $(\mathbb{Z}, +)$ is a group, so every element $x \in \mathbb{Z}$ has an inverse $-x$.

Another useful property for a semigroup is idempotence. A semigroup $(\mathbf{S}, +)$ is *idempotent* if $x + x = x$ for all $x \in \mathbf{S}$. The *idempotent version* often admits more efficient solutions, compared to both the semigroup and group versions, because the same element can be counted multiple times without affecting the result. An important special case is *range emptiness*, which can be modeled by the boolean idempotent semigroup $(\{0, 1\}, \vee)$ and assigning

$w(p) = 1$ for all $p \in P$.

In order to design a range searching data structure, we need to know the range space \mathcal{R} from which the query ranges are drawn. Some examples of range spaces \mathcal{R} examined in this dissertation are:

- *Convex ranges*: The set of all d -dimensional convex shapes.
- *Orthogonal ranges*: The set of all d -dimensional axis-aligned rectangles.
- *Halfspace ranges*: The set of all d -dimensional halfspaces.
- *Spherical ranges*: The set of all d -dimensional balls.
- *Smooth ranges*: The set of all convex shapes such that every point in the boundary of the shape can be touched by a ball of constant radius contained inside the shape.
- *Simplex ranges*: The set of all d -dimensional simplices, where a *simplex* is defined as the (possibly unbounded) intersection of $d + 1$ halfspaces.
- *Fat simplex ranges*: The set of all d -dimensional simplices formed by the intersection of $d + 1$ halfspaces such that the angle between the boundary of any two halfspaces is at least a constant α .

We should note that some of the range spaces mentioned above are subsets of other range spaces. For example, spherical ranges are a special type of convex ranges. Not surprisingly, spherical range searching can be solved more efficiently than convex range searching. Also, halfspace ranges are limiting cases of both simplex and spherical ranges. A diagram relating the different range spaces is presented in Figure 1.2.

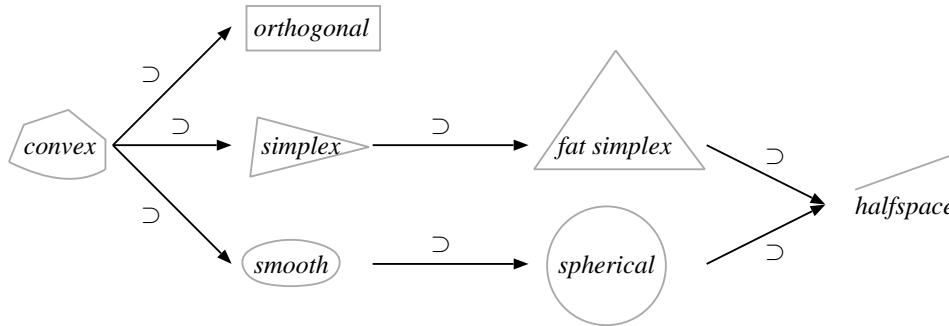


Figure 1.2: Subset relation between different range spaces.

The relatively high complexity of exact range searching has lead researchers to consider the problem in the context of approximation. There are several ways to define approximation for the range searching problem. In the range counting problem, one can define approximation in terms of the result of the query function. There are two natural ways to approximate the counting. Let $\varepsilon > 0$ be an approximation parameter, and $q_\varepsilon(R)$ be the result of an ε -approximate query. With *relative counting error*, we have

$$(1 - \varepsilon)q(R) \leq q_\varepsilon(R) \leq (1 + \varepsilon)q(R).$$

Approximate range counting with relative counting error has been studied in [1, 5]. With *absolute counting error*, we have

$$\frac{q(R)}{|P|} - \varepsilon \leq \frac{q_\varepsilon(R)}{|P|} \leq \frac{q(R)}{|P|} + \varepsilon.$$

Approximate range searching with absolute counting error have been extensively studied under the topic of ε -approximations [25, 27, 55] (see Section 3.3.2 for more details).

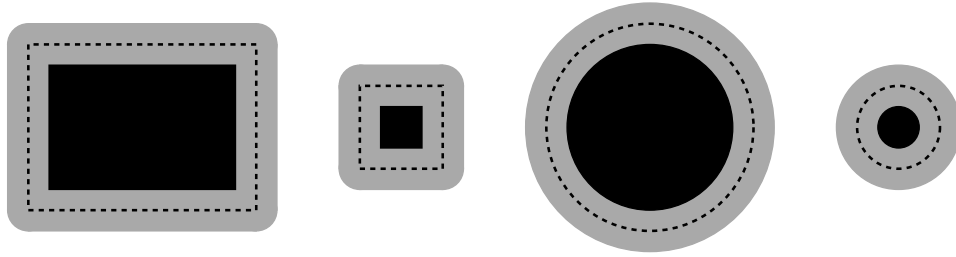


Figure 1.3: Fuzzy boundaries in the absolute error model.

A drawback of approximating the count is that it does not generalize to arbitrary semigroups, since a notion of proximity between elements of the semigroup is required. A more geometric way to define approximation is to consider the range boundary to be “fuzzy,” and allow points that are close to the range boundary to either be counted or not. This approach generalizes to arbitrary semigroups because it considers proximity in the space where the points are embedded, instead of proximity between the semigroup elements. There are two natural ways in which to define this kind of approximation. In both cases a user-supplied approximation parameter $\varepsilon > 0$ is given. In the *relative error model* (or simply *relative model* for short) it is assumed that the range shape R is bounded, and points lying within distance $\varepsilon \cdot \text{diam}(R)$ of the boundary of the range may or may not be included.

In contrast, in the *absolute error model* (or simply *absolute model*) points lying within distance ε of the range boundary may or may not be included, regardless of the diameter of the range, as illustrated in Figure 1.3. The region within distance ε of the range boundary is called the *fuzzy boundary*. In the absolute error model, the thickness of the fuzzy boundary is independent of the diameter of the range.

Note that, in the absolute model, some type of scaling is needed. Oth-

erwise, it would be possible to answer queries with arbitrarily high precision by applying some high scale factor to the point and range coordinates, while keeping the error parameter fixed. Without loss of generality, we assume throughout that the point set P has been transformed (through a uniform scaling and translation) to lie within the unit hypercube $[0, 1]^d$. We assume that the ranges have been similarly transformed, and the parameter ε has been scaled correspondingly.

Approximate range searching in the relative model has been studied in [8, 9, 10, 12]. Chazelle, Liu and Magen [26] studied approximate range searching in the absolute model, but considered the problem in spaces of high dimension. They presented a data structure which answers halfspace queries in $O((d/\varepsilon)^2 \log^{O(1)}(d/\varepsilon))$ time with $dn^{O(1/\varepsilon^2)}$ storage. Throughout this dissertation, we assume that the dimension d is constant.

There are a number of reasons for studying approximate range searching in the absolute model. First, the absolute model admits much simpler solutions. While the most efficient data structures for answering range queries both in the relative model and in the exact case tend to be quite complex, our techniques are extremely simple (involving simple structures such as grids and quadtrees) and so are amenable to efficient implementation. In the absolute model, the data structures are not sensitive to the point distribution. Therefore, the absolute model allows us to reason about the range searching problem (for example, the best size and shape of the generators) in a simpler context, and may lead to more efficient and simpler structures for exact range searching.

Second, the absolute model is better suited for several applications, when compared to the relative model. If the coordinates of a point represent an

object that exists within some extent in space, or data that is subject to measurement errors or noise, then the approximation quality should be based on the expected error of the point locations, not on the diameter of the query range. Another shortcoming of the relative model is that it cannot meaningfully handle unbounded ranges, such as halfspaces and unbounded polyhedra, because the allowable error increases with the diameter of the range.

Finally, the storage space and query time of the absolute model data structures is independent of n , and hence our results can be adapted to work in the *data stream model* [14, 48]. In the data stream model, the data set is too large to fit in memory. Therefore, the storage space should be independent of n (sometimes polylogarithmic functions of n are acceptable). Also, the data points are examined one at a time, in a single pass, while queries regarding the points that have already been seen need to be answered efficiently. Exact range searching clearly requires $\Omega(n)$ storage for all reasonable range spaces, and approximate range searching in the relative model requires $\Omega(n)$ storage when the query ranges can be scaled arbitrarily. Suri, Tóth, and Zhou [53] consider approximate range counting in the data stream model, approximating the number of points inside the query region.

1.1 Summary of Results

The main results of this dissertation include the introduction of the absolute model, approximate range searching data structures for several range spaces in the absolute model, and application of these data structures to related problems.

This dissertation is the first systematic work on approximate range searching in the absolute model. The absolute model represents a natural way to define geometric approximation. The absolute model is particularly suited for practical problems where some form of approximation is performed when determining the coordinates of the data points. Possible sources of such approximations are measurements errors, rounding, and noise. Another source is the fact that points are often used to approximate objects with extent in space or the expected values of data which is probabilistic in nature.

We develop data structures for the most fundamental shapes of ranges: orthogonal regions, convex regions, halfspaces, Euclidean balls, and simplices. The data structures are significantly more efficient and simpler than their exact counterparts, and therefore are amenable to efficient implementation. Several data structures involve a space-time tradeoff, where the query time can be reduced at the cost of increasing the storage space. We also present data structures that benefit from specific properties of the semigroup, such as idempotence and existence of inverse. These data structures are also described in [33].

We apply the techniques developed for the absolute model to several other problems, including exact range searching, approximate range searching in the relative model, approximate nearest neighbor searching, and the data stream model. Perhaps the most notable application is approximate range searching in the relative model, where we develop a data structure which is not only simpler but also has reduced query time, storage space, and preprocessing time when compared to previous data structures. We also describe this relative model data structure in [34].

Range	Version	Storage space	Query time	Preprocessing	Sec.
Convex	semigroup	$O(1/\varepsilon^d)$	$O(1/\varepsilon^{d-1})$	$O(n + 1/\varepsilon^d)$	4.2
Orthogonal	semigroup group	$m \geq 1/\varepsilon^d$	$O(\alpha(m, 1/\varepsilon^d))$	$O(n + m)$	4.1
		$O(1/\varepsilon^d)$	$O(1)$	$O(n + 1/\varepsilon^d)$	4.1
Halfspace	semigroup idempotent emptiness	$O(1/\varepsilon^d)$	$O(1)$	$\tilde{O}(n + 1/\varepsilon^d)$	5.2
		$m \geq 1/\varepsilon^{(d+1)/2}$	$O(1/m\varepsilon^d)$	$\tilde{O}(n + 1/\varepsilon^d)$	5.3
		$m \geq 1/\varepsilon^{(d-1)/2}$	$O(1/m\varepsilon^{d-1})$	$O(n + 1/\varepsilon^d)$	5.4
Spherical	semigroup semigroup emptiness	$\tilde{O}(1/\varepsilon^d)$	$O(1/\varepsilon^{(d-1)/2})$	$\tilde{O}(n + 1/\varepsilon^d)$	6.2
		$m \geq 1/\varepsilon^{d+1}$	$O\left(1/m^{\frac{1}{2}} - O(\frac{1}{d})\varepsilon^{d-O(1)}\right)$	$O\left(n + m/\varepsilon^{\frac{d-1}{2}}\right)$	6.2
		$O(1/\varepsilon^d)$	$O(1/\varepsilon^{(d-1)/2})$	$O(n + 1/\varepsilon^d)$	6.3
Simplex	group	$\tilde{O}(1/\varepsilon^d)$	$O(1/\varepsilon^{d-2} + \log(1/\varepsilon))$	$\tilde{O}(n + 1/\varepsilon^d)$	6.4
Fat simplex	semigroup	$\tilde{O}(1/\varepsilon^d)$	$O(1/\varepsilon^{d-2} + \log(1/\varepsilon))$	$\tilde{O}(n + 1/\varepsilon^d)$	6.4

Table 1.1: Complexities of several approximate range searching data structures.

Because most of our results make use of bucketing, we assume a model of computation that supports integer division. We use $\tilde{O}(x)$ to denote $O(x \log^{O(1)} x)$, $\tilde{\Omega}(x)$ to denote $\Omega(x/\log^{O(1)} x)$, $\alpha(m, n)$ to denote the inverse Ackermann function [54], and \log to denote the base-2 logarithm. We also assume that $d \geq 2$ unless otherwise specified. The complexities of our data structures are summarized in Table 1.1. We describe the data structures in more detail below.

- *Orthogonal ranges (semigroup and group versions)*: In Section 4.1, we present a reduction from approximate orthogonal range searching to the partial sum problem. The technique consists of approximating the data points using a grid aligned set of points. Then, we convert the set of grid aligned points into a multidimensional array and use existing partial sum data structures to answer orthogonal range queries.
- *Convex ranges (semigroup version)*: In Section 4.2, we explain how to use a quadtree to answer range queries for arbitrary convex ranges. We

assume that some information regarding the intersection of the range with a hypercube can be computed in constant time.

- *Halfspace ranges (semigroup version)*: In Section 5.2, we show how to obtain an approximate halfspace range searching data structure with constant query time and $O(1/\varepsilon^d)$ storage space. We also describe how to preprocess the data structure efficiently. This data structure is an important component of the halfbox quadtree introduced later.
- *Halfspace ranges (idempotent version)*: In Section 5.3, we show how to obtain a space-time tradeoff for the idempotent version of halfspace range searching. The technique consists of properly distributing a set of large Euclidean balls. The tradeoff is obtained by varying the diameter and the number of balls. In the high-storage end of the tradeoff, the balls degenerate to halfspaces, and we obtain the semigroup data structure, with constant query time.
- *Halfspace ranges (emptiness version)*: In Section 5.4, we explain how to compress the idempotent halfspace range searching data structure for the emptiness case, reducing the storage space by a factor of $1/\varepsilon$ without affecting the query time.
- *Halfbox quadtree*: In Section 6.1, we introduce the halfbox quadtree, which efficiently answers approximate range queries for several range shapes. The data structure combines the quadtree from Section 4.2 with the approximate halfspace range searching data structure from Sections 5.2 or 5.4. The result is a data structure that allows the arbitrary cutting

angles provided by halfspaces and the varying box sizes provided by the quadtree.

- *Spherical ranges (semigroup and emptiness versions)*: In Sections 6.2 and 6.3, we show how to use the halfbox quadtree to answer spherical range queries. The idea is to approximate the boundary of a sphere using flat regions bounded by quadtree boxes. Our results apply not only to spherical ranges, but also to arbitrary smooth ranges. We also show how to improve the spherical query time at the cost of additional space (this result does not apply to general smooth ranges).
- *Simplex ranges (semigroup and group versions)*: In Section 6.4, we show how to use the halfbox quadtree to answer simplex range queries. The general case requires the use of subtraction, and therefore only applies to the group version. If the query simplex is fat, the use of subtraction is not necessary.

Another important part of our work consists of applying approximate range searching data structures developed for the absolute model to related problems. Several applications are explained below.

- *Range sketching*: In Section 4.3, we introduce the range sketching problem, which tries to fill the gap between range counting and range reporting. We present a quadtree based data structure for answering range sketching queries with arbitrary ranges.
- *Exact idempotent halfspace range searching*: In Section 5.5, we apply the approximate halfspace range searching data structure from Section 5.3

to answer exact halfspace range searching queries in the idempotent version. This exact data structure is defined in the semigroup arithmetic model [10, 18, 22], and we assume that the data points are uniformly distributed inside the unit hypercube. The data structure has $O(n^{1-2/(d+1)})$ expected query time with $O(n)$ space, matching the lower bound proved in [18] up to logarithmic factors. The theoretical importance of the data structure relies on the fact that uniform distribution and the semigroup arithmetic model are also assumed in the lower bound proved in [18]. Therefore, we open some important theoretical and practical questions: Is the average case complexity for uniformly distributed data strictly lower than the worst case complexity? Does the semigroup arithmetic model allow more efficient idempotent halfspace range searching data structures than the real RAM model?

- *Approximate nearest neighbor:* In Section 6.3, we show how to use $O(\log(1/\varepsilon))$ approximate spherical emptiness queries to answer approximate nearest neighbor queries in the absolute error model.
- *Range reporting:* In Section 6.5, we show how to modify the halfbox quadtree to answer range reporting queries efficiently.
- *Data stream model:* In the *data stream model* [14, 48], the data set is too large to fit in memory, and the data points are examined one at a time, in a single pass, while queries regarding the points that have already been seen need to be answered efficiently. In Section 6.6, we show how to modify the halfbox quadtree in order to make it more efficient for the

data stream model. Our results are presented in the form of a query time versus update time tradeoff.

- *Relative model:* In Section 6.7, we combine a compressed quadtree, a finger tree, and the absolute model approximate range searching data structure from Section 5.2 to obtain a data structure analogous to the halfbox quadtree, but suited for the relative error model. We show how to use the data structure to answer smooth range queries and simplex range queries in the relative model. Our results are presented in the form of space-time tradeoffs, and are an improvement over some of the best results previously known.

1.2 Organization

This dissertation is divided into 7 Chapters, as follows.

In Chapter 1, we define approximate range searching, explain the importance of studying approximate range searching in the absolute error model, and summarize the results contained in this dissertation.

In Chapter 2, we define terms and discuss previous results that are used throughout this dissertation. We define the models of computation that we use, introduce the concept of partition trees, and prove several geometric lemmas.

In Chapter 3, we summarize previous work on range searching. The results explained in this chapter are important for comparison purposes, but are not strictly necessary for understanding our results. Most of the chapter is devoted to exact range searching. Several exact range searching data structures and

techniques are presented.

In Chapter 4, we develop approximate range searching data structures for two types of ranges: orthogonal ranges and convex ranges. We also introduce the range sketching problem, which tries to fill the gap between range counting and range reporting. A data structure for range sketching with arbitrary ranges is presented.

In Chapter 5, we present approximate halfspace range searching data structures for the semigroup, idempotent, and emptiness versions of the problem. We also apply the results to obtain an exact halfspace range searching data structure for the idempotent version of the problem, in the semigroup arithmetic model and assuming uniform point distribution.

In Chapter 6, we introduce a data structure called the *halfbox quadtree*. The halfbox quadtree is our most versatile data structure. It combines a quadtree with approximate halfspace range searching data structures to obtain an efficient data structure for various range shapes, including spherical and simplex ranges. We present variations of the halfbox quadtree suited for approximate nearest neighbor queries, approximate range reporting queries, the data stream model, and the relative error model.

Finally, in Chapter 7, we present conclusions and directions for future work.

Chapter 2

Preliminaries

In this chapter, we define terms and discuss results that will be used throughout this dissertation. In Section 2.1, we formalize the absolute error model, by defining the ASA model, which is used to describe our data structures. In Section 2.2, we introduce the concept of partition trees, and more specifically quadtrees. In Section 2.3, we state and prove some geometric lemmas that are used throughout this dissertation.

2.1 The ASA Model

Given a range $R \in \mathcal{R}$ and an approximation parameter $\varepsilon > 0$, we define R^+ as the locus of points x such that $\text{dist}(x, R) \leq \varepsilon$. We define R^- as the locus of points x such that $\text{dist}(x, \bar{R}) \geq \varepsilon$, where \bar{R} is the complement of R . We say that R_ε ε -approximates R within B if

$$(R^- \cap B) \subseteq (R_\varepsilon \cap B) \subseteq (R^+ \cap B).$$

We say that R_ε ε -approximates R if R_ε ε -approximates R within $[0, 1]^d$. We say $q_\varepsilon(R)$ is an ε -approximation of $q(R)$ if there is R_ε such that $q_\varepsilon(R) = q(R_\varepsilon)$ and R_ε approximates R .

We define a computational model, called the *Approximate Semigroup Arithmetic model* (*ASA model* for short), which makes it easier to describe our data structures. The ASA model is similar to the semigroup arithmetic model [10, 18, 22]. We explain how to convert our approximate data structures from the ASA model to the real RAM model (with integer division), preserving the same query time and storage space.

Given a collection of sets \mathcal{S} , let $\bigcup(\mathcal{S}) = \bigcup_{S \in \mathcal{S}} S$. In the *semigroup version*, we say that a set of regions \mathcal{G} and a function $g : \mathcal{R} \rightarrow 2^{\mathcal{G}}$ ε -generates \mathcal{R} if, for all $R \in \mathcal{R}$, the sets in $g(R)$ are pairwise disjoint, and $\bigcup(g(R))$ ε -approximates R . The elements of \mathcal{G} are called *generators*. In the *idempotent version* the elements of $g(R)$ do not need to be pairwise disjoint, because $x + x = x$ for all $x \in \mathbf{S}$. In the *group version*, as the elements of the semigroup $(\mathbf{S}, +)$ have an inverse, the generators can be summed and subtracted in a multiset fashion, as long as the final result is a set, that is, no point is counted more than once, or counted a negative number of times. For simplicity, we use $\bigcup(g(R))$ to refer to the sums and subtractions of generators in the group version.

Our definition of generators is different than the standard semigroup arithmetic model definition. We define a generator as a region of the space. In the semigroup arithmetic model, a generator is defined to be a linear form of weights of a set of data points. Our definition is more natural for the data structures described in this paper, and can be generalized to handle non-discrete point sets.

In the ASA model, a set \mathcal{G} and a function g that ε -generates \mathcal{R} is called a *data structure* for \mathcal{R} . The *storage space* of the data structure is defined as $|\mathcal{G}|$, and the *query time* is defined as $T(\mathcal{G}, \mathcal{R}) = \max_{R \in \mathcal{R}} |g(R)|$. We say that a data structure provides *internal approximation* if $\bigcup(g(R)) \subseteq R$, for all $R \in \mathcal{R}$, and provides *external approximation* if $R \subseteq \bigcup(g(R))$, for all $R \in \mathcal{R}$. Modifying our data structures to provide either internal approximation or external approximation is straightforward.

There would be little practical value to develop upper bounds using the ASA model, if we could not convert the data structures from the ASA model to a more standard model of computation such as the real RAM model. The ASA model (as well as the semigroup arithmetic model) considers neither preprocessing time nor the time to identify the proper generators for a given range. Identifying the proper generators consists of computing $g(R)$ efficiently, and is a simple task for the data structures we present, with the exception of the exact halfspace data structure from Section 5.5.

Preprocessing consists of computing $w(G) = \sum_{p \in P \cap G} w(p)$, for all $G \in \mathcal{G}$. We may modify our set of generators \mathcal{G} in order to obtain a set that is faster to preprocess, by using an approximation of each $G \in \mathcal{G}$, instead of G itself, when computing $w(G)$. We provide details on how to preprocess our data structures efficiently throughout the text. Therefore, our data structures work in the real RAM model, with the exception of the exact halfspace data structure from Section 5.5, which works in the semigroup arithmetic model.

2.2 Partition Trees and Quadtrees

Several kinds of partition trees (defined below) have application to range searching. In particular, several new results presented in this dissertation are based on the quadtree, which is a particular type of partition tree. Other partition trees are used in exact range searching, and approximate range searching in the relative model. We discuss the quadtree in this section, and discuss several other partition trees in Chapter 3.

A *partition tree* is a hierarchical subdivision of the set of data points P . Each node v in the partition tree is associated with a *cell* v_{\square} . The root of the partition tree is associated with some kind of bounding box for the set of data points. Each internal node v has children c_1, \dots, c_k such that $\bigcup_{i=1}^k c_{i\square} \subseteq v_{\square}$. Each node v is also associated, implicitly or explicitly, with a set of data points $\dot{v} \subseteq v_{\square} \cap P$. The root node v_0 is associated with $\dot{v}_0 = P$. If c_1, \dots, c_k are the children of a node v , then $\dot{c} = \bigcup_{i=1}^k \dot{c}_i$ and $\dot{c}_i \cap \dot{c}_j = \emptyset$ for $i \neq j$. The sum of the weights of the data points in \dot{v} , denoted by $w(v) = \sum_{p \in \dot{v}} w(p)$, is precomputed and associated with node v .

First, we introduce the *partition tree query algorithm*, which answers a query $q(R)$ through a standard recursive approach. The query starts by calling $q(R, v_0)$, where v_0 is the root of the partition tree. The procedure $q(R, v)$ considers four cases in order:

1. If $v_{\square} \cap R = \emptyset$, then return 0.
2. If $v_{\square} \cap R = v_{\square}$, then return the precomputed $w(v)$.
3. If v is a leaf node, then return $\sum_{p \in \dot{v} \cap R} w(p)$.

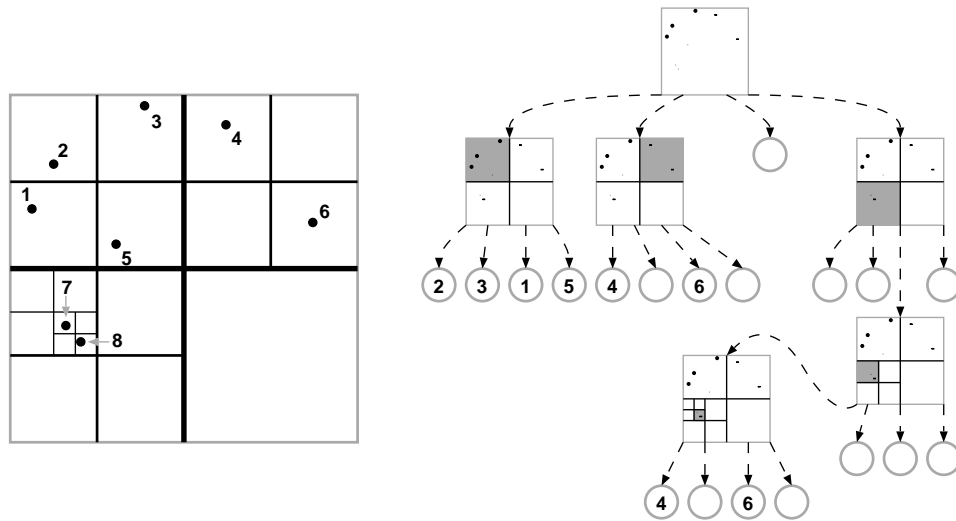


Figure 2.1: Representation of a quadtree.

4. Otherwise, recursively return $\sum_c q(R, c)$, for all children c of v .

It is easy to see that the algorithm above correctly answers a range query. The query time is proportional to the number of nodes visited during the execution of the query algorithm, which depends on the specific partition tree, as well as the shape of the range.

The most natural partition tree is probably the *region quadtree*, or just *quadtree*. In a quadtree, the cell v_{\square} associated with a node v is a d -dimensional hypercube. Each internal node has 2^d children corresponding to the disjoint subdivisions of v_{\square} by d hyperplanes (perpendicular to each orthogonal axis) that divide v_{\square} into equal cells. Since the subdivisions of a node are disjoint, we have $\dot{v} = v_{\square} \cap P$. Generally, a node v is a leaf node if $|\dot{v}| \leq 1$, but different criteria can be defined to stop the subdivision process. A quadtree is represented in Figure 2.1. See the book by Samet [52] for a detailed discussion on quadtrees.

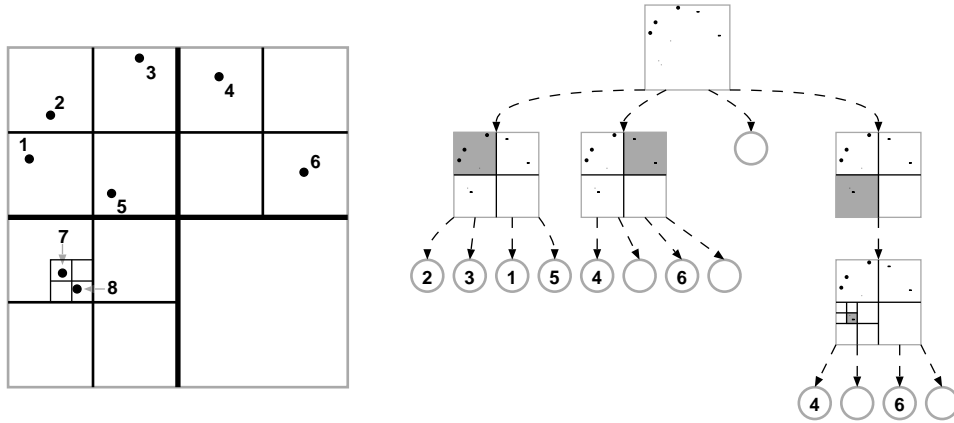


Figure 2.2: Representation of a compressed quadtree.

A *quadtree box* is defined recursively as the original bounding hypercube or the hypercubes obtained by evenly dividing a quadtree box by d hyperplanes perpendicular to each orthogonal axis. Quadtrees are not efficient in the worst case, because both the size and the depth of a quadtree are unbounded with respect to n . To see this intuitively, imagine the case when the set P consists of only two points that are very close together in a somewhat random position.

There are different variations of the quadtree, that remove certain unnecessary nodes to save space and reduce query time. In the *compressed quadtree* [32, 35], we replace 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 $O(n)$, but compressed quadtrees are still inefficient in the worst case, as the depth of the tree can be as high as $\Theta(n)$. Nevertheless, the depth of a compressed quadtree is $O(\log n)$ for several natural point distributions. A compressed quadtree is represented in Figure 2.2. A compressed quadtree storing n points can be built in $O(n \log n)$ time [13].

The worst-case $\Theta(n)$ depth of the compressed quadtree makes it inefficient for answering queries. There are several alternatives to remedy that, like the BBD-tree [12, 13] and the skip quadtree [32]. In this dissertation, we use a somewhat simpler approach which is based on [35].

A *separator* for a tree T with n nodes is a node v such that removing v from T produces a forest F where every tree in F have at most $n/2$ nodes.

Lemma 2.1. *Every tree T with n nodes has a separator, and it can be computed in $O(n)$ time.*

Proof. Consider the following algorithm, started with v as the root of the tree. We assume that the number $D(v)$ of descendants of a node v , including v itself, is precomputed and stored in the tree.

1. If $D(v) < n/2$, then return the parent of v .
2. Otherwise, find the child v' of v that maximizes $D(v')$ and call the algorithm recursively for v' .

The algorithm descends through a path v_1, \dots, v_{k+1} in T , with $D(v_1) = n$ and $D(v_i) > D(v_{i+1})$, and returns v_k as a separator. To show that v_k is actually a separator, we need to show that all trees obtained by removing v_k from v have less than $n/2$ nodes. The subtrees rooted at the children of v_k have less than $n/2$ nodes because the largest of these trees, the one rooted at v_{k+1} , has $D(v_{k+1}) < n/2$ nodes. Also, since $D(v) \geq n/2$, the tree obtained by removing from T the subtree rooted at v has less than $n/2$ nodes. \square

We build the *finger tree* T' by setting the root of T' to be a separator v of T , and the children of v to be the roots of the finger trees of the trees obtained

by removing v from T . Since we can find a separator in $O(n)$ time, and the depth of the finger tree is $O(\log n)$, the total time to build a finger tree is $O(n \log n)$. A node v in the finger tree T' can have two types of children. A *parent-type* child is a child u of v in T' such that u is a parent of v in T . A *descendant-type* child is a child u of v in T' such that u is a descendant of v in T . A node v can have up to 2^d descendant-type children, but at most 1 parent-type children.

A important type of query that can be answered efficiently with the use of a finger tree is called a cell query. Given a query quadtree box Q , a *cell query* consists of finding the only quadtree box Q' in T such that $P \cap Q = P \cap Q'$, if it exists. To understanding the meaning of a cell query, we examine the different possible results. If Q is in T , then we have $Q' = Q$. Else, if $P \subset Q$, then $Q' \supset Q$ is the root of the quadtree. Otherwise, Q' is the largest quadtree box in T such that $Q' \subset Q$. The only case in which Q' does not exist is when $Q' \cap P = \emptyset$, in which case we return 0.

To answer a cell query $q(Q, v)$, we start with v as the root of T' , and apply the following procedure. The procedure takes $O(\log n)$ time because it only descends in the finger tree, and the depth of the finger tree is $O(\log n)$.

1. If $Q = v_{\square}$, then return v .
2. If $Q \supset v_{\square}$, then determine the parent-type child c of v . If v has no parent-type child or $c_{\square} \supset Q$, then return v . Otherwise, recursively return $q(Q, c)$
3. If $Q \subset v_{\square}$, then determine the child-type child c of v such that $c_{\square} \subset Q$. If c does not exist, then return 0. Otherwise, recursively return $q(Q, c)$.

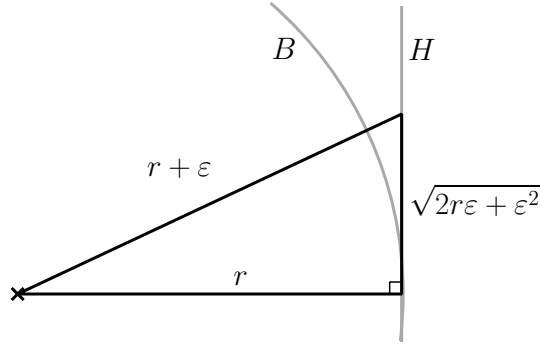


Figure 2.3: Triangle used in the proof of Lemma 2.2.

2.3 Some Geometric Lemmas

In this section, we define and prove some geometric lemmas that are used throughout this dissertation. The first lemma shows how well spherical surfaces and hyperplanes approximate each other. It is used in Section 5.3, when a halfspace is approximated by balls, and also in Section 6.2, when a ball is approximated by flat regions.

Lemma 2.2. *Let B be a d -dimensional ball of radius r , and H be a hyperplane that is tangent to B at point p . Consider the $(d - 1)$ -dimensional ball $B' \subset H$ of radius $\sqrt{2r\epsilon + \epsilon^2} > \sqrt{r\epsilon}$, centered at p . Then, all points in B' are within distance ϵ from B .*

Proof. The proof follows from applying the Pythagorean Theorem on the triangle formed by two rays of B and a segment of length $\sqrt{2r\epsilon + \epsilon^2}$ connecting the two points in the intersection of the two rays with H (Figure 2.3). \square

The remaining lemmas are all *packing lemmas*, that is, they limit the number of disjoint objects that can intersect some region, as a function of the size of the objects and the region. In our case, the objects are always quadtree boxes

(defined in Section 2.2). The following packing lemma follows from Lemma 2 in [12].

Lemma 2.3. *If \mathcal{Q} is a set of pairwise disjoint quadtree boxes, each of diameter at least δ , that intersect a region R of diameter $\Delta \geq \delta$, then*

$$|\mathcal{Q}| = O\left(\left(\frac{\Delta}{\delta}\right)^d\right).$$

Proof. Consider the smallest quadtree box B that contains R . The diameter of B is $O(\Delta)$, and the volume of B is $O(\Delta^d)$. Since two quadtree boxes can only intersect when one is completely contained inside the other, we can divide the volumes of B and the quadtree boxes in \mathcal{Q} to bound

$$|\mathcal{Q}| = \frac{O(\Delta^d)}{O(\delta^d)} = O\left(\left(\frac{\Delta}{\delta}\right)^d\right).$$

□

We can improve the previous result when the region consists of the $(d-1)$ -dimensional boundary of a convex d -dimensional region. The following packing lemma follows from Lemma 3 in [12].

Lemma 2.4. *If \mathcal{Q} is a set of pairwise disjoint quadtree boxes, each of diameter at least δ , that intersect the boundary of a convex region R of diameter $\Delta \geq \delta$, then*

$$|\mathcal{Q}| = O\left(\left(\frac{\Delta}{\delta}\right)^{d-1}\right).$$

Proof. Let ∂R denote the boundary of R . Note that we can consider only quadtree boxes of diameter exactly δ , since a larger quadtree box can be

replaced by several smaller boxes, out of which at least one intersects the boundary of R . Therefore, we consider a grid of cells of diameter δ , and need to count how many grid cells intersect ∂R .

We partition ∂R according to the normal vectors. The *normal vector* of a point $p \in \partial R$ is the unit vector perpendicular to ∂R at point p . If there are multiple normal vectors, pick an arbitrary one. Partition the points in ∂R according to which face of a hypercube centered at the origin the normal vector points to. Since a d -dimensional hypercube has $2d$ faces, we are partitioning ∂R in $2d$ components. Without loss of generality, we prove that the lemma holds for the points in one partition S of ∂R . We say that S corresponds to the top face of the hypercube. We define the direction of the normal vector of that face as the vertical direction, and the horizontal $(d - 1)$ -dimensional hyperplane as perpendicular to the vertical direction.

We claim that S cannot intersect all grid cells in a vertical stack of more than $m = O(1)$ quadtree boxes. From Lemma 2.3 applied to the $(d - 1)$ -dimensional projection of S , we have that S can only intersect $O((\Delta/\delta)^{d-1})$ parallel vertical stacks of quadtree boxes of diameter at least δ . Therefore, the total number of quadtree boxes of diameter at least δ that intersects ∂R is $O((\Delta/\delta)^{d-1})$.

To prove the claim, consider a stack of m grid cells of diameter δ . The height of the stack is $m\delta/\sqrt{d}$, and to intersect all quadtree boxes in the stack R needs to have points with vertical distance at least $(m - 2)\delta/\sqrt{d}$. The horizontal distance inside the stack can be at most $\delta\sqrt{d - 1}/\sqrt{d}$. Since S bounds a convex region and has normal pointing to the top face of the hypercube, we

have

$$\frac{\frac{(m-2)\delta}{\sqrt{d}}}{\frac{\delta\sqrt{d-1}}{\sqrt{d}}} \leq \frac{\sqrt{d}}{\sqrt{d-1}},$$

which implies that $m \leq \sqrt{d} + 2$. \square

We define a $(d-2)$ -flat region as a subset of a $(d-2)$ -dimensional hyperplane. We can still improve the previous result when the region consists of $(d-2)$ -flats.

Lemma 2.5. *If \mathcal{Q} is a set of pairwise disjoint quadtree boxes, each of diameter at least δ , that intersect the $(d-2)$ -faces of a simplex of diameter $\Delta \geq \delta$, then*

$$|\mathcal{Q}| = O\left(\left(\frac{\Delta}{\delta}\right)^{d-2}\right).$$

Proof. A d -dimensional simplex has $\binom{d+1}{d-1} = O(1)$ number of $(d-2)$ -faces, all of which are $(d-2)$ -flat. Consequently, it is sufficient to show that a $(d-2)$ -flat region R of diameter $\Delta > \delta$ can only intersect $O((\Delta/\delta)^{d-2})$ quadtree boxes.

As in the proof of Lemma 2.4, we consider a grid of cells of diameter δ , and need to count how many grid cells intersect R . Without loss of generality, we assume that R is a subset of a hyperplane H such that the normal vector of H points to the top face of a hypercube centered at the origin.

Then, we can use the same argument as in the proof of Lemma 2.4, to show that R can only intersect $O(1)$ grid cells in each stack, and then use Lemma 2.4 on the projection of R onto the horizontal $(d-1)$ -dimensional hyperplane to obtain the desired $O((\Delta/\delta)^{d-2})$ bound. \square

A shape R is α -fat [50] if, for all d -dimensional balls B with center in R

and not fully containing R , we have

$$\alpha \text{ volume}(R \cap B) \geq \text{volume}(B).$$

A shape is *fat* if there is some constant α for which it is α -fat. If a simplex is fat, then the angle between any two $(d - 1)$ -dimensional faces is bounded by a constant. The following packing lemma considers the intersection between quadtree boxes and strictly more than one face of a fat simplex.

Lemma 2.6. *If \mathcal{Q} is a set of pairwise disjoint quadtree boxes, each of diameter exactly δ and intersecting at least two $(d - 1)$ -faces of a fat simplex R of diameter $\Delta \geq \delta$, then*

$$|\mathcal{Q}| = O\left(\left(\frac{\Delta}{\delta}\right)^{d-2}\right).$$

Proof. Consider two $(d - 1)$ -faces F_1, F_2 of R . Let E be the $(d - 2)$ -face $F_1 \cap F_2$. A quadtree box of diameter δ that intersects both F_1 and F_2 must be within distance $O(\delta)$ from E , since the angle between the two $(d - 1)$ -hyperplanes containing each of F_1 and F_2 is greater than a constant. Let S be the set of points of H within distance δ from E .

Even though the region S has diameter $\Theta(\Delta)$ (because E itself has diameter $\Theta(\Delta)$), the width of S with respect to any direction perpendicular to E is only $O(\delta)$. We can partition S using a $(d - 2)$ -dimensional grid that partitions E into cells of diameter δ (and extends infinitely in the other two dimensions), obtaining $O((\Delta/\delta)^{d-2})$ smaller regions of diameter δ . Applying Lemma 2.3 to each of these smaller regions, we conclude that $O(1)$ quadtree boxes intersect each smaller region. Therefore, the total number of quadtree boxes that inter-

sect S (and the number of quadtree boxes that simultaneously intersects F_1 and F_2) is $O((\Delta/\delta)^{d-2})$. □

Chapter 3

Previous Work

In this chapter, we summarize previous work on exact range searching, and approximate range searching in the relative error model. The results explained in this chapter are important for comparison purposes, but are not strictly necessary for understanding the new results introduced in this dissertation. We present data structures and techniques for exact orthogonal range searching in Section 3.1. Since the data structures for exact simplex range searching and exact halfspace range searching are mostly similar, we handle these two range spaces together in Sections 3.2 and 3.3. In Section 3.2, we discuss the case of linear space data structures, and in Section 3.3, we discuss the case of polylogarithmic query time. Finally, in Section 3.4 we discuss approximate range searching in the relative error model.

3.1 Orthogonal Range Searching

The exact data structures for orthogonal range searching are much more efficient than the exact data structures for any other reasonable range. Many exact structures have polylogarithmic query time with linear or near-linear storage. Some of these structures are close to the lower bound of $\Omega(\log(n/\log(2m/n))^{d-1})$ query time for $O(m)$ space in the semigroup arithmetic model [23]. For a good survey, see [2]. We discuss two simple data structures for orthogonal range searching: kd-trees and range trees.

3.1.1 kd-Trees

The simplest partition tree that guarantees $O(\log n)$ depth is the *kd-tree* [16, 29]. In a kd-tree, the cell v_{\square} associated with a node v is a d -dimensional axis-aligned rectangle. Each internal node has two children c_1 and c_2 corresponding to the disjoint subdivisions formed by a splitting v_{\square} with a hyperplane perpendicular to an orthogonal axis a . Several different criteria are used to define the split direction for each vertex of the kd-tree. The standard criterion consists of cycling through the different axis in an arbitrary, but constant, order. Since the subdivisions of a node are disjoint, we have $\dot{v} = v_{\square} \cap P$. The splitting hyperplane is chosen in a way that $|\dot{c}_1|$ and $|\dot{c}_2|$ differ by at most 1. A vertex v such that $|\dot{v}| \leq 1$ is a leaf node in the tree. A kd-tree uses $O(n)$ storage space and can be constructed in $O(n \log n)$ preprocessing time. A 2-dimensional kd-tree and the associated spatial subdivision is represented in Figure 3.1.

To analyze the query time, we break the execution tree of the query algorithm into sequences of d levels of recursive calls, one for each different split

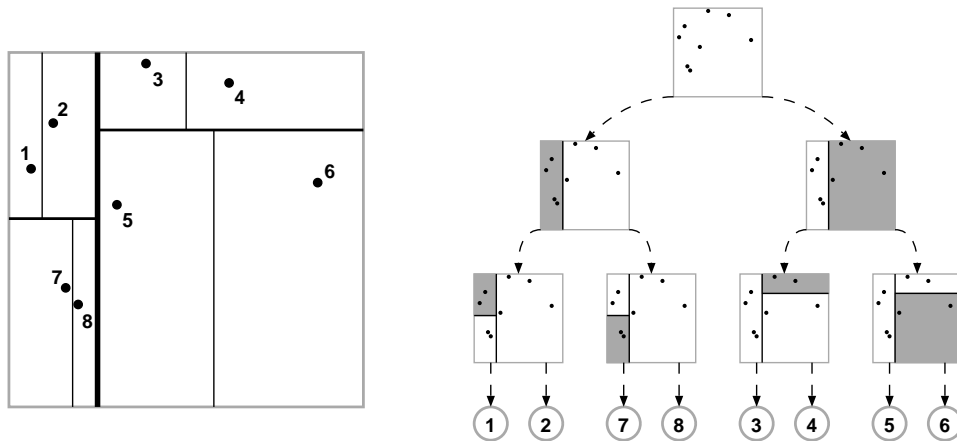


Figure 3.1: Representation of a 2-dimensional kd-tree.

direction. First, we consider the case where the query range is an axis-aligned halfspace. Since the halfspace is axis aligned, there is one splitting direction that is parallel to the halfspace boundary. Therefore, only one recursive call is performed for each query in the corresponding level of the tree. We can write down the following recurrence for the query time:

$$T(n) = 2^{d-1}T(n/2^d).$$

The recurrence solves to $T(n) = O(n^{1-1/d})$. To analyze the number of recursive calls performed when answering an orthogonal range query, we note that if a vertex of the kd-tree is visited when answering an orthogonal range query, then it is also visited when answering a query for one of its bounding halfspaces. Therefore, the kd-tree can answer orthogonal range queries in $O(n^{1-1/d})$ time, with $O(n)$ storage space, and $O(n \log n)$ preprocessing time.

3.1.2 Range Trees

The first orthogonal range searching data structure to achieve polylogarithmic query time with almost linear space is the *range tree*. Range trees were independently discovered by several authors [17, 37, 39, 56]. Our presentation is based on [29]. We start by describing the 1-dimensional version, and then generalize the construction to higher dimensions.

In 1-dimensional space, all convex ranges are intervals of the real line. To answer orthogonal range searching in 1-dimensional space, we can use a variation of a binary search tree. The data points are stored in the leaf nodes. The *key* of a leaf node is the coordinate of the (1-dimensional) point stored in it, and the *weight* of a leaf node is the weight of the point stored in it. The *weight* of an internal node is recursively defined as the sum of the weights of its children. The *key* v_k of an internal node v is a real value such that, the key of all nodes in the left subtree of v is less than or equal to v_k , and the key of all nodes in the right subtree of v is greater than v_k . In order to support range searching efficiently, the tree needs to be balanced, that is, the height of a tree storing n elements should be $O(\log n)$. A 1-dimensional range tree is represented in Figure 3.2.

To answer a range query $q(x_1, x_2, v)$ for the interval $[x_1, x_2]$, we use the following recursive algorithm, starting with v as the root of the tree. Let v_k , v_w , v_l , and v_r denote the key, weight, left child, and right child of node v , respectively.

1. If $|x_1| = |x_2| = \infty$, then return v_w .
2. If $v_k \in [x_1, x_2]$, then return $q(x_1, \infty, v_l) + q(-\infty, x_2, v_r)$.

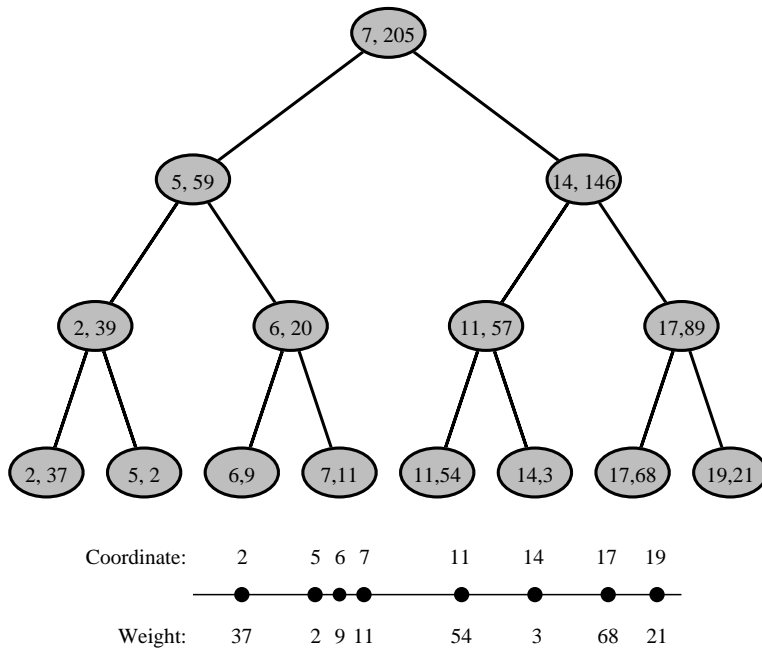


Figure 3.2: Representation of a 1-dimensional range tree.

3. If $v_k < x_1$, then return $q(x_1, x_2, v_r)$.
4. If $v_k > x_1$, then return $q(x_1, x_2, v_l)$.

To calculate the query time, we need to note that when case (2) happens, the recursive calls have ∞ as at least one range boundary. After one of the range boundaries is set to ∞ , one of the recursive calls performed in (2) takes $O(1)$ time, since it goes to case (1). Therefore, the maximum query time is proportional to the height of the tree, which is $O(\log n)$.

In order to deal with more than one dimensions, we use the multi-level data structure technique. A *multi-level data structure* is formed by data structures that, instead of returning a set of points as the answer to a query, returns a set of data structures for the corresponding set of points. A 2-dimensional range tree is a 1-dimensional range tree on the x coordinate of the points where the

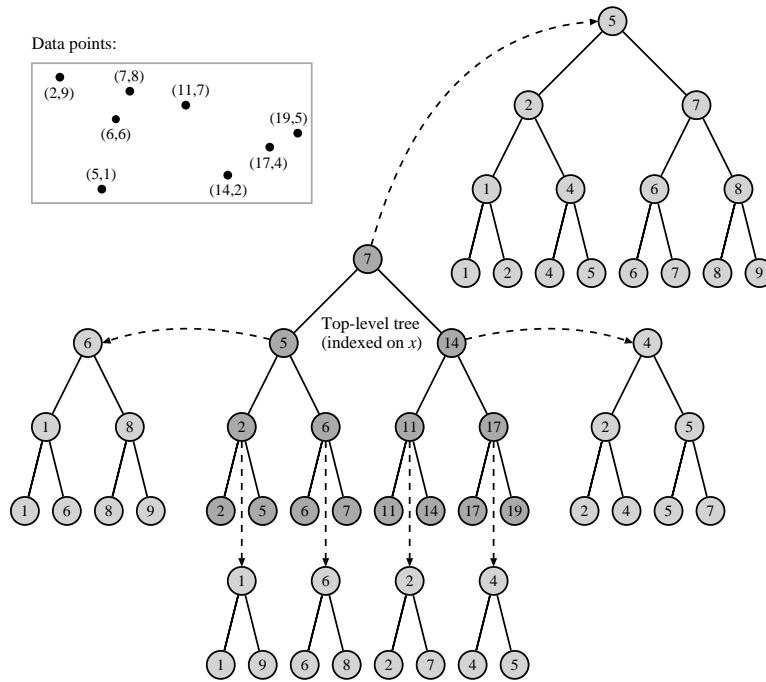


Figure 3.3: Representation of a 2-dimensional range tree. Point weights are not represented.

weight v_w of an internal node v is the 1-dimensional range tree storing the elements in the subtree rooted at v and indexed by the y coordinate of the points (Figure 3.3). The storage space is $O(n \log n)$. The orthogonal range query time is $O(\log^2 n)$, since the 1-dimensional query algorithm needs to be executed for each coordinate. The same approach can be used to build d -dimensional range trees with $O(n \log^{d-1} n)$ storage space, and $O(\log^d n)$ query time for orthogonal range queries.

Chazelle [20, 21] used compressed range trees and other techniques to improve the storage space and query of range trees, obtaining some of the most efficient orthogonal range searching data structures known, for different models of computation. Chazelle's results for the real RAM are summarized in

Range	Version	Storage space	Query time	Preprocessing
orthogonal	semigroup	$O(n \log^{d-2} n)$	$O(\log^{d+\varepsilon} n)$	$O(n \log^{d-1} n)$
orthogonal	semigroup	$O(n \log^{d-2} n \log \log n)$	$O(\log^d n \log \log n)$	$O(n \log^{d-1} n)$
orthogonal	semigroup	$O(n \log^{d-2+\varepsilon} n)$	$O(\log^d n)$	$O(n \log^{d-1} n)$
orthogonal	group	$O(n)$	$O(\log^{d-1} n)$	$O(n \log^{d-1} n)$

Table 3.1: Complexities of some of the most efficient exact orthogonal range searching data structures. In this table, ε represents an arbitrarily small constant.

Table 3.1.

3.2 Simplex and Halfspace Range Searching with Linear Space

In simplex range searching, the ranges are d -dimensional simplices. If m units of storage are allowed, then the query time is $\Omega(n/\sqrt{m})$ in the plane, and $\Omega((n/\log n)/m^{\frac{1}{d}})$ in d -dimensional space [22]. In the exact version, the most efficient linear size data structure is due to Matoušek [42] and matches the lower bounds up to logarithmic factors, achieving $O(n^{1-\frac{1}{d}})$ query time for the case of linear storage space.

In Section 3.2.1, we present a simple data structure based on ham sandwich cuts. In Section 3.2.2, we introduce the concept of simplicial partitions, which is used in several of the most efficient data structures known.

3.2.1 Ham Sandwich Cut

The classic *ham sandwich cut* problem (see [31]) is:

Problem 3.1. *Given two disjoint sets of points P_1, P_2 in the plane, find a*

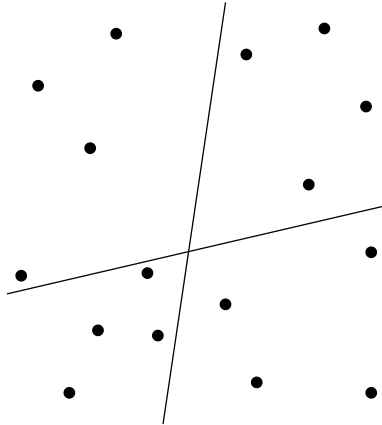


Figure 3.4: Set of 16 points divided into four subsets of the same size by 2 lines.

halfplane h such that $|P_1 \cap h| = \lfloor |P_1|/2 \rfloor$ and $|P_2 \cap h| = \lfloor |P_2|/2 \rfloor$.

It is not immediately clear that h exists. Not only does h always exist, but it can also be found in linear time [38]. In the case when P_1 and P_2 are separated by a line, Megiddo gives a simpler linear-time algorithm in [47]. The ham sandwich cut existence implies that we can partition a set of points by two lines in a way that each of the four remaining sets have the same number of points (or a difference of at most 1 point), as shown in Figure 3.4. We can divide the points recursively using this process, until we have at most one point per cell.

To analyze the query time for a halfplane range query in the partition tree described above, we note that a halfplane can intersect at most 3 out of the 4 regions in each subdivision. As the number of points is the same in all 4 regions, we have the following recurrence relation for the query time:

$$T(n) \leq O(1) + 3T(n/4).$$

Solving the recurrence, we obtain $T(n) = O(n^{\log_4 3}) = O(n^{0.792})$. This is

significantly sublinear, but still far from the optimum $O(\sqrt{n})$ query time. Note that the same data structure can answer simplex queries (triangle queries, in the case) with the same query time, because every node visited when answering a simplex query is also visited when answering a query for one of the three halfplanes that define the simplex.

3.2.2 Simplicial Partitions

Let P be a set of n points in d -dimensional space, and r be a parameter with $2 \leq r \leq n/2$. A *simplicial partition* of size r , is a collection of r d -dimensional simplices $\Delta_1, \dots, \Delta_r$. Each simplex Δ_i is associated with a set of points $P_i \subseteq \Delta_i \cap P$, and each point $p \in P$ is associated with a simplex in the collection. Differently than in the partition trees described before, the simplices may and often will intersect each other. On the other hand, the sets of points associated with any two different simplices are still required to be disjoint, that is, $P_i \cap P_j = \emptyset$ for $i \neq j$. An example of simplicial partition is in Figure 3.5.

A simplicial partition is *fine* if $|P_i| \leq 2n/r$ for $1 \leq i \leq r$. The *crossing number* κ of a simplicial partition is the maximum number of cells that a hyperplane can intersect. The following theorem is proved in [40]:

Theorem 3.2. *For any set P of n points in d -dimensional space ($d \geq 2$), and any parameter r with $2 \leq r \leq n$, there exists a fine simplicial partition of size r and crossing number $\kappa = O(r^{1-1/d})$.*

We can use a fine simplicial partition to produce a partition tree. We start by building a fine simplicial partition for P and recursively build a partition

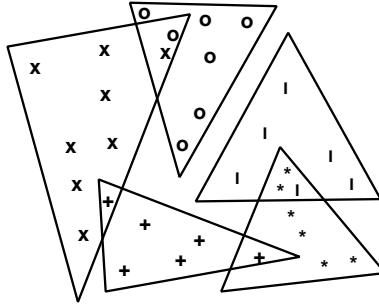


Figure 3.5: Example of a simplicial partition in the plane. The points are drawn with different shapes according to the different sets P_i .

tree for each set P_i in the simplicial partition. Each node in the tree has r children, the height of the tree is $O(\log_r n)$, and the number of nodes is $O(n)$.

A halfspace range query (or simplex range query) can be answered using the standard partition tree query algorithm from Section 2.2. The query time is defined by the following recurrence with base case $T(O(1)) = O(1)$:

$$T(n) \leq O(r) + \kappa T(2n/r) = O(r) + O(r^{1-1/d}) T(2n/r).$$

If we set $r = n^\varepsilon$, where ε is a sufficiently small constant, then the query time is $O(n^{1-1/d} \log^{O(1)} n)$.

3.3 Simplex and Halfspace Range Searching in Polylogarithmic Time

Matoušek [42] showed that it is possible to answer a halfspace range query in $O(\log n)$ time with $O((n/\log n)^d)$ storage space and a simplex range query in $O(\log^{d+1} n)$ time with $O(n^d)$ storage space. The data structures are strongly

based on Chazelle’s hierarchical cuttings [24] (Section 3.3.3). For the case of simplex range searching, a multi-level data structure is used. In Section 3.3.1, we show how to reformulate halfspace range searching by using geometric duality. In Section 3.3.2, we introduce a more abstract view of range searching. In Section 3.3.3, we introduce the concept of cuttings.

3.3.1 Geometric Duality

In general, an operation is *dual* if recursively applying the operation to an object twice produces the same original object. The most common duality in geometry consists of mapping points to hyperplanes and hyperplanes to points, in a way that incidence and order properties are preserved. One way to define a duality transform in the plane consists of mapping a point (a, b) into a line $y = ax - b$, and also mapping a line $y = ax - b$ into the point (a, b) . In d -dimensional space, we can map a point (a_1, \dots, a_d) to a line $x_d = a_1x_1 + \dots + a_{d-1}x_{d-1} - a_d$. We represent the dual of a point p by p^* and the dual of a hyperplane h by h^* . We say that the original points and lines are in the *primal space*, while their duals are in the *dual space*. An example of primal and dual spaces is in Figure 3.6. More details can be found in [29]. The following properties hold:

Property 3.3. *Incidence preserving: $p \in h$ if and only if $h^* \in p^*$.*

Property 3.4. *Order preserving: p lies above h if and only if h^* lies above p^* .*

Halfspace range searching can easily be formulated in the dual plane:

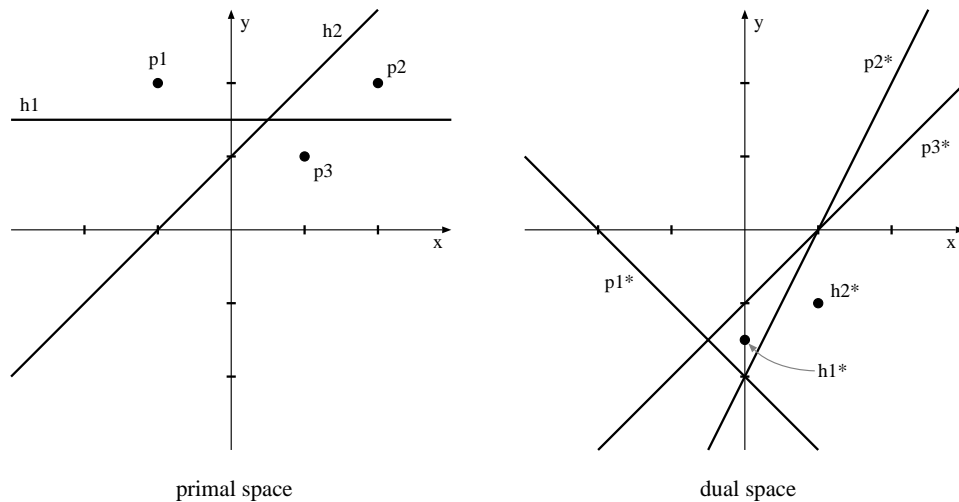


Figure 3.6: Example of primal and dual spaces. The dual of the point (a, b) is the line $y = ax - b$.

Problem 3.5. Let H be a set of n hyperplanes in \mathbb{R}^d equipped with weights. Preprocess H in a way that, given a query point q , we can find the sum of weights of the hyperplanes lying above (or below) q .

Problem 3.5 can be solved as a point location problem in a d -dimensional arrangement of n hyperplanes, using the hierarchical cuttings described in Section 3.3.3. The data structure obtained this way has $O(\log n)$ query time with $O(n^d)$ storage space.

3.3.2 VC-Dimension, ε -Nets, and ε -Approximations

In this section, we examine range searching in a more abstract, combinatorial setting. A *set system* (P, \mathcal{R}) is formed by a universe set P and a set \mathcal{R} containing subsets of P . In the range searching setting, the set P corresponds to the set of n points, and the set \mathcal{R} corresponds to the intersection of P with each range. We call \mathcal{R} a *range space*. Sometimes the sets P and \mathcal{R} are infinite.

A subset $S \subseteq P$ is called an ε -net if $S \cap R \neq \emptyset$ for every $R \in \mathcal{R}$ with $|R|/|P| > \varepsilon$. A subset $A \subseteq P$ is called an ε -approximation if it satisfies the following stronger property:

$$\left| \frac{|A \cap R|}{|A|} - \frac{|R|}{|P|} \right| \leq \varepsilon.$$

Intuitively, an ε -approximation is a subset of P that approximates P with respect to the fraction of the points that intersects each range in \mathcal{R} . An ε -net is a subset of P that approximates P with respect to emptiness queries, where ranges with too few points may be misclassified as empty. Surprisingly, it is possible to obtain both ε -nets and ε -approximations with size independent of n for several important range spaces.

We cannot expect ε -nets and ε -approximations with size independent of n to exist for any set system. A sufficient condition for the existence of such small ε -nets and ε -approximations is bounded VC-dimension. The *VC-dimension* of a set system is the size of the largest subset $X \subseteq P$ such that, for all $X' \subseteq X$, there is a set $R \in \mathcal{R}$ with $R \cap X = X'$ [55]. A set system has *bounded VC-dimension* if its VC-dimension is finite. For example, it is an easy exercise to show that $(\mathbb{R}^2, \mathcal{R})$, where \mathcal{R} is the set of all halfplanes, has VC-dimension 3.

Set systems of bounded VC-dimension not only have an ε -net of size $O(1/\varepsilon \log(1/\varepsilon))$, but a random sample of this size (with a large enough multiplicative constant that depends on the VC-dimension) is an ε -net with a positive probability. Similarly, set systems of bounded VC-dimension have an ε -approximation of size $O(1/\varepsilon^2 \log(1/\varepsilon))$, and a random sample of this size is an ε -approximation with high probability.

Both an ε -net and an ε -approximation of optimal sizes can be computed deterministically in $O(n/\varepsilon^{O(1)})$ time, where the $O(1)$ in the exponent depends on the VC-dimension of the set system [44]. The deterministic computation of ε -nets and ε -approximations is crucial for the derandomization of most geometric algorithms [45].

3.3.3 Cuttings

Let H be a set of n hyperplanes in d -dimensional space. A partition of \mathbb{R}^d into simplices is called an ε -cutting for H if each simplex intersects at most εn hyperplanes of H . Cuttings can be used to design geometric divide and conquer algorithms.

A *canonical triangulation* of an arrangement is a triangulation obtained with the following recursive procedure. The 1-faces of the arrangement are already triangulated. To triangulate a k -face F , connect the lowest vertex of F (the one with minimum x_d coordinate) to each simplex obtained in the triangulation of the $(k - 1)$ -faces that bound F .

It is possible to construct ε -cuttings with only $O(1/\varepsilon^d)$ simplices. An initial, and almost successful, approach to compute an ε -cutting is the following: Pick a random subset X of c/ε^d hyperplanes of H , for some sufficiently large constant c . Then, compute a canonical triangulation T of the arrangement produced by H , and return T .

Unfortunately, the algorithm in the previous paragraph does not generally produce an ε -cutting. Nevertheless, the simplices of T intersect at most $O(\varepsilon n \log(1/\varepsilon))$ hyperplanes of H with a constant positive probability. To re-

move the undesired $\log(1/\varepsilon)$ factor, we need to refine T using a second step of sampling.

For each simplex $t \in T$, compute the set I_t of hyperplanes of H that intersect t . If $|I_t| \leq \varepsilon n$, we keep the simplex t . Otherwise, we randomly sample $c|I_t|/\varepsilon n \log(|I_t|/\varepsilon n)$ halfspaces of I_t , for a sufficiently large constant c , and replace t by a canonical triangulation of the intersection of t and the random sample. We repeat this procedure until T is an ε -cutting. The expected number of repetitions is $O(1)$.

A weakness of cuttings generated this way, is that they do not necessarily compose well, as we show in this paragraph. Imagine we construct an ε^k -cutting X_k (for some integer value k that is not necessarily asymptotic constant) by building an ε -cutting X_1 , and recursively building a cutting X_i for the halfspaces intersected by each simplex in X_{i-1} , for i from 2 to k . Unfortunately, the size of X_k is not $O(1/\varepsilon^{dk})$, but really $O(c^k/\varepsilon^{dk})$. The reason is that the constant hidden in the O notation accumulates through the k levels of the recursion.

Fortunately, Chazelle [24] gives a deterministic algorithm to produce a hierarchy of cuttings X_1, X_k , where X_i is an ε^i -cutting obtained by refining the simplices of X_{i-1} , and the size of each cutting X_i is $O(1/\varepsilon^{di})$. Such a hierarchy of cuttings is called a *hierarchical cutting*, and it has several applications, the most straightforward one being point location in an arrangement of n hyperplanes with $O(n^d)$ storage space and $O(\log n)$ query time.

Using geometric duality, point location in hyperplane arrangements is equivalent to halfspace range searching. By using some additional techniques, the storage space of halfspace range searching can be reduced to $O((n/\log n)^d)$

while keeping the $O(\log n)$ query time.

3.4 Relative Error Model

In the *relative error model* (or simply *relative model*), it is assumed that the range shape R is bounded, and points lying within distance $\varepsilon \cdot \text{diam}(R)$ of the boundary of the range may or may not be included. Range searching in the relative model has been studied in [8, 9, 10, 12].

The BBD-Tree is the simplest data structure proposed for approximate range searching in the relative model. The BBD-tree requires $O(n)$ storage space and $O(n \log n)$ preprocessing time. Using the BBD-tree, we can answer approximate range queries in $O(\log n + 1/\varepsilon^d)$ time for arbitrary ranges and $O(\log n + 1/\varepsilon^{d-1})$ time for convex ranges [12]. In both cases, we used the following *unit-cost test assumption*: given a range R and a d -dimensional hypercube, in constant time we can determine whether the hypercube is contained within R , is disjoint of R , or neither.

The cells of the BBD-tree are the set difference between two quadtree boxes. The BBD-tree has two types of nodes: *split nodes* and *shrink nodes*. A split node is similar to the typical quadtree subdivision. In a split node, an axis aligned halfspace is used to divide a cell through its midpoint, splitting the cell in its longest orthogonal direction. A shrink node v corresponding to a cell v_\square has two children corresponding to the cells u_\square and $v_\square \setminus u_\square$, where $u_\square \subset v_\square$ is a quadtree box. An example of a BBD-tree is illustrated in Figure 3.7.

The BBD-tree is also a very important tool for storing approximate Voronoi diagrams [6, 7], which solve the approximate nearest neighbor problem. The

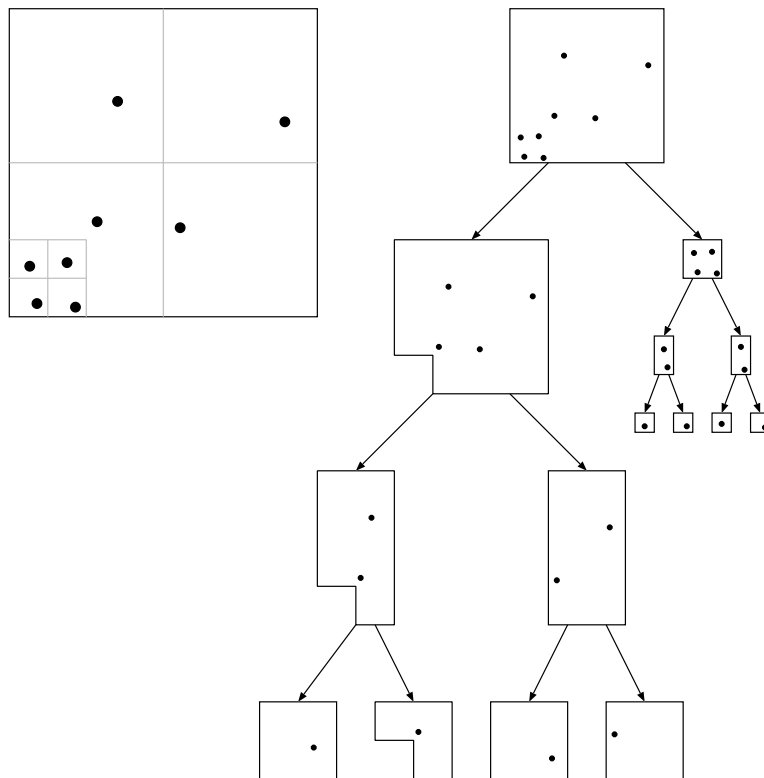


Figure 3.7: Example of a BBD-tree.

nearest neighbor problem consists of preprocessing a set P of n data points in d -dimensional space in a way that, given a query point p_q (generally not in P), we can efficiently find the point $p^* \in P$ that minimizes $\text{dist}(p_q, p^*)$. In the relative model, the *approximate nearest neighbor problem* consists of preprocessing a set of data points P in a way that, given a query point p_q , a point $p' \in P$ such that $\text{dist}(p_q, p') \leq (1 + \varepsilon)\text{dist}(p_q, p^*)$ can be computed efficiently. An *approximate Voronoi diagram* is a subdivision of the space in a way that, for every query point p_q , the leaf cell v such that $p_q \in v_\square$ is associated with a set of data points \dot{v} (\dot{v} does not need to be a subset of v_\square in this case) with $p' \in \dot{v}$. If $|\dot{v}| = 1$ for every leaf node, then the approximate Voronoi diagram is said to have a *single-representative*. If $|\dot{v}| \geq 1$ for some leaf node, then the approximate Voronoi diagram is said to have *multiple-representatives*. In [7], Arya, Malamatos and Mount present an approximate Voronoi diagram with multiple representatives that solves the approximate nearest neighbor problem in $O(\log n + 1/\varepsilon^{(d-1)/2})$ query time with $O(n)$ space, as well as space time tradeoffs.

Arya, Malamatos, and Mount [8] introduced a space-time trade-off for approximate spherical range searching, which has faster query time than the standard BBD-tree, at the cost of additional storage space. Given a parameter $\gamma \in [1, 1/\varepsilon]$, an approximate spherical range query can be answered in $O(\log(n\gamma) + 1/(\varepsilon\gamma)^{d-1})$ time with $O(n\gamma^d \log(1/\varepsilon))$ storage space and $O(n\gamma^d \log(n/\varepsilon) \log(1/\varepsilon))$ preprocessing time.

A semigroup is *idempotent* if $x + x = x$ for all semigroup elements x . Arya, Malamatos, and Mount [10] introduced a space-time trade-off for approximate spherical range searching under idempotent semigroups. Given a

parameter $\gamma \in [1, 1/\varepsilon]$, an approximate spherical range query can be answered in $O(\log(n) + 1/(\varepsilon\gamma)^{(d-1)/2} \log(1/\varepsilon))$ time with $O(n\gamma^d/\varepsilon)$ storage space and $O(n(\gamma/\varepsilon)^{(d+1)/2} \log(n/\varepsilon))$ preprocessing time. If the range is an arbitrary smooth range and the unit cost test assumption is used, then approximate range queries under idempotent semigroups can be answered in $O(\log n + 1/\varepsilon^{(d-1)/2})$ time with $O(n/\varepsilon)$ storage space [9].

There are several lower bounds for approximate range searching in the relative model. Arya and Mount [12] showed that any partition tree takes $\Omega(\log n + 1/\varepsilon^{d-1})$ time to answer orthogonal hypercube queries, and therefore the query time of the BBD-tree for arbitrary convex ranges is optimal. In the semigroup arithmetic model with m units of storage, approximate spherical range queries take $\tilde{\Omega}\left(\left(\frac{1}{\varepsilon}\right)^{\frac{d}{2}-1} \left(\frac{n}{m}\right)^{\frac{1}{2}-\frac{1}{2(d+1)}}\right)$ time [10]. In the semigroup arithmetic model with $m = n(1/\varepsilon)^{f^2 d^2}$ units of storage, approximate rotated unit hypercube range range queries take $\Omega\left(\left(\frac{1}{\varepsilon}\right)^{d-2\sqrt{d}-2fd}\right)$ time.

A semigroup $(\mathbf{S}, +)$ is *integral* if $kx \neq x$ for all $k \in \mathbb{N}^+$ and $x \in \mathbf{S} \setminus \{0\}$. If we assume that the semigroup is integral and the generators are convex, then several tighter lower bounds for approximate range searching in the relative model exist. Approximate spherical range queries take $\tilde{\Omega}\left(\left(\frac{1}{\varepsilon}\right)^{d-5} \left(\frac{n}{m}\right)^{1-\frac{4}{d}}\right)$ time with m units of storage [10]. Approximate rotated unit hypercube range range queries take $\Omega\left(\left(\frac{1}{\varepsilon}\right)^{d-2-2fd}\right)$ time with $m = n(1/\varepsilon)^{f^2 d^2}$ units of storage.

Chapter 4

Orthogonal and Convex Ranges

In this chapter, we develop approximate range searching data structures for two types of ranges: orthogonal ranges and convex ranges. We also introduce the range sketching problem.

In Section 4.1, we introduce orthogonal range searching data structures, for both the group and semigroup versions of the problem. The data structures are based on a simple point approximation technique.

In Section 4.2, we introduce convex range searching data structures. The data structures for convex range searching are based on a quadtree decomposition of space, which is also used in the halfbox quadtree (Chapter 6).

In Section 4.3, we introduce the range sketching problems and data structures for solving it. The solution is based on compressed quadtrees and finger trees, in a manner similar to the relative error model data structures presented in Section 6.7.

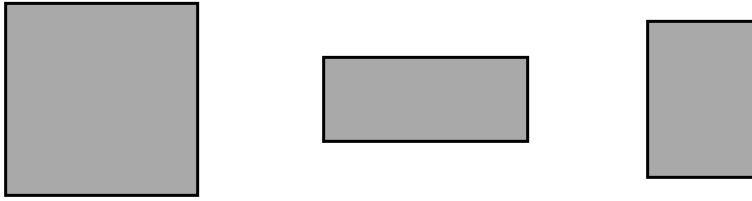


Figure 4.1: Examples of orthogonal ranges.

4.1 Orthogonal Range Searching

In the orthogonal range searching problem, \mathcal{R} is the set of all axis-parallel rectangles. Some examples of orthogonal ranges are shown in Figure 4.1. In this section, we reduce approximate orthogonal range searching in the absolute model to the partial sums problem.

The exact data structures for orthogonal range searching are much more efficient than the exact data structures for any other reasonable range. Many exact structures have polylogarithmic query time with linear or near-linear storage. Some of these structures are close to the lower bound of $\Omega(\log(n/\log(2m/n))^{d-1})$ query time for $O(m)$ space in the semigroup arithmetic model [23]. For a good survey, see [2].

In [12], Arya and Mount prove that if the points are stored in any partition tree, the worst case complexity of answering approximate range counting queries for axis-parallel unit hypercube ranges is $\Omega(1/\varepsilon^{d-1})$. The lower bound also holds for the absolute model because the proof of [12] involves only ranges of unit size. We show that we can do much better by not using partition trees.

The *point approximation* technique consists of creating a new weighted set of points P' , in a way that approximate range queries on P are equivalent to exact range queries on P' . The concept is somewhat similar to ε -approximations,

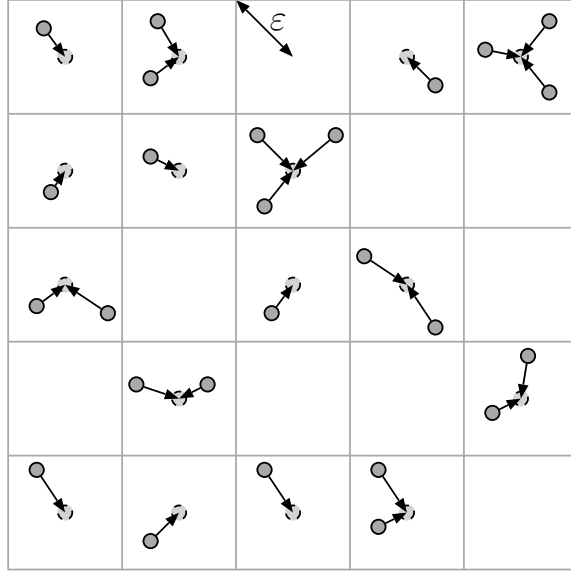


Figure 4.2: Point approximation.

except that P' is generally not a subset of P .

To build P' , we partition the space using a rectangular grid with cells of diameter 2ε . We bucket the points from P into the grid cells and, for each non-empty grid cell, we create a new point $p \in P'$ located in the center of the grid cell (Figure 4.2). The weight of p is defined as the sum of the weights of all points of P contained in the same grid cell as p . Since $P \subset [0, 1]^d$, we have $|P'| \leq \min(n, O(1/\varepsilon^d))$.

The set P' can be converted into a d -dimensional array A with $O(1/\varepsilon^d)$ elements, where $A(a_1, \dots, a_d) = w(p)$ and $p \in P'$ is the point within the grid cell a_1, \dots, a_d , as shown in Figure 4.3. Consequently, an orthogonal range searching query is equivalent to a d -dimensional interval query in A . We denote the size of each dimension of the array A by $u = O(1/\varepsilon)$.

The problem of answering a d -dimensional interval query in an array has been well studied before [28, 57], and is called the *partial sum* problem.

1 •	0	0	• 2 •	↖ ε 0
0	• 1	1	• • 3 •	1 •
• 1	• 2 •	• 1	• 1	0
0	• 4 •	• 2	0	• 1
• 1	1 •	0	• • 3 •	0

Figure 4.3: Array built for a set of points with weight 1 in the plane.

4.1.1 Group Version

In this section, we study the group version of the partial sum problem. We consider an d -dimensional array A where each dimension has size u and the total size is u^d . In the group version, it is possible to achieve constant query time with linear storage.

Given $X = (x_1, \dots, x_d)$ and $Y = (y_1, \dots, y_d)$, we represent the portion of the array A formed by the elements between $A(X)$ and $A(Y)$, including both $A(X)$ and $A(Y)$, by $A(X..Y)$. We also represent the “unbounded” interval $A(X..(u, \dots, u))$ by $A(X..\infty)$.

Let $q(X, Y)$ denote the query for the range $A(X..Y)$, and $q(X)$ denote the query for the range $A(X..\infty)$. As there are only $O(1/u^d)$ grid cells, we can build a lookup table for $q(X)$ using $O(u^d)$ space. This table can be built in constant time per element using the principle of inclusion-exclusion:

$$q(X) = A(X) - \sum_{Z \in \{(z_1, \dots, z_d) : z_i \in \{x_i, x_i+1\}\} \setminus \{X\}} (-1)^{\delta(Z, X)} q(Z),$$

where $\delta(Z, X)$ is the number of indices i such that $z_i \neq x_i$.

Bounded rectangular queries can be answered using $2^d = O(1)$ unbounded queries, since there is a subtraction operation and d is constant. We can rewrite a bounded query as

$$q(X, Y) = \sum_{Z = (z_1, \dots, z_d) : z_i \in \{x_i, y_i\}} (-1)^{\delta(Z, X)} q(Z).$$

The following theorem summarizes the main results of this section:

Theorem 4.1. *There exists an ε -approximate range searching data structure for orthogonal ranges with $O(1/\varepsilon^d)$ size, $O(n + 1/\varepsilon^d)$ preprocessing time, and $O(1)$ query time, for the group version of the problem.*

4.1.2 Semigroup Version

We start this section by describing two simple solutions to the partial sum problem, in the semigroup version. The first one gives an approximate range searching data structure with $O(1/\varepsilon^d)$ space and $O(\log^d(1/\varepsilon))$ query time. The second one gives an approximate range searching data structure with $O(1/\varepsilon^d \log(1/\varepsilon))$ space and $O(1)$ query time. Then, we present the complexities of the optimal data structures [28, 57], which gives an approximate range searching data structure with $O(m)$ space and $O(\alpha(m, n)^d)$ query time, for $m \geq 1/\varepsilon^d$.

For simplicity, we assume that u , the size of each dimension of the array, is

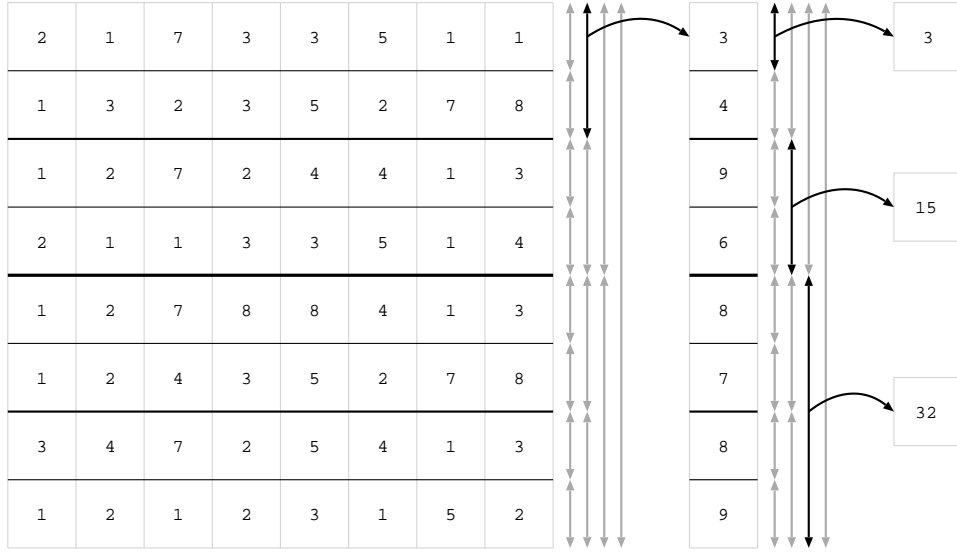


Figure 4.4: Subdividing an array recursively in a fixed dimension and linking to a $(d - 1)$ -dimensional data structure. Only parts of the data structure are shown.

a power of 2. The first data structure we present is similar to range trees [29], but optimized for the case of arrays.

We can build a tree by recursively subdividing the array in half, splitting in a fixed dimension. We repeat this process until we cannot divide the array any more, as in the left Figure 4.4. Then, for each subdivision, we calculate the sums in the dimensions different than the one we just subdivide, producing a $(d - 1)$ -dimensional array, as in Figure 4.4. We then build the same data structure for the $(d - 1)$ -dimensional array, repeating this process until $d = 0$. The idea of linking an element in a data structure to another data structure is called a *multi-level data structure* [29].

The total size of all the k -dimensional data structures is $O(u^d)$. As k goes from 0 to d , the total size of the structure is $O(u^d)$. A query is performed in each dimension, by finding the largest intervals that add together to the total

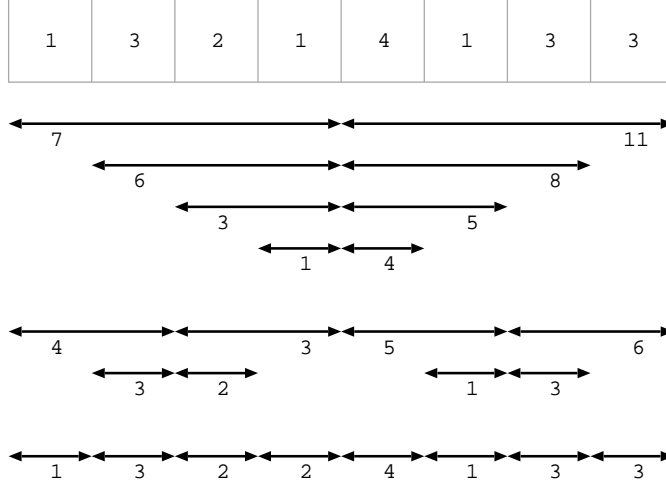


Figure 4.5: Data structure with $O(1)$ query time for the semigroup version of partial sum.

query range in that dimension, and recursively answering the queries for the lower dimension data structures. The query time is $O(\log^d(u))$.

We restrict our presentation of the second data structure to the case when $d = 1$. Extending the result to multi-dimensional spaces is simple and can be done using multi-level data structures, as in the previous example. Consider a single-dimensional array with elements (v_1, \dots, v_w) . We can precompute and store $\sum_{i=k}^{u/2} v_i$, for $1 \leq k \leq u/2$ and $\sum_{i=u/2+1}^k v_i$, for $u/2 + 1 \leq k \leq u$. We can answer any query that includes elements in both halves of the array in $O(1)$ time by adding two precomputed values. A query that only includes element in one half of the array can be answered by another data structure built the same way for the given half of the array (Figure 4.5). The total size for all the data structures is $O(u \log u)$, and the query time is $O(1)$.

In the d -dimensional version, the storage space becomes $O(u^d \log u)$ and the query time remains $O(1)$.

An optimal static partial sum data structures are due to Chazelle and Yao [28, 57]. The data structures have $O(\alpha(m, u^d))$ query time with $O(m)$ storage space, where $m \geq u^d$. The function $\alpha(m, n)$ is the inverse Ackermann function defined in [54]. According to the value of m , the data structure allows constant query time with slightly superlinear space or linear space with very slow growing query time.

The following theorem summarizes the main results of this section:

Theorem 4.2. *There exists an ε -approximate range searching data structure for orthogonal ranges with $O(m)$ size (for $m \geq 1/\varepsilon^d$), $O(n + m)$ preprocessing time, and $O(\alpha(m, 1/\varepsilon^d)^d)$ query time, for the semigroup version of the problem.*

4.1.3 Reporting Version

Overmars [49] present two data structures to answer orthogonal range reporting when the data points are on a $u \times u$ grid in the plane. Let k denote the number of points reported in a query. The first data structure has $O(\sqrt{\log u} + k)$ query time with $O(n \log n)$ storage space and preprocessing time. The second data structure has $O(\log \log u + k)$ query time with $O(n \log n)$ storage space and $O(u^3 \log u)$ preprocessing time.

Using these structures on the set P' , we obtain ε -approximate orthogonal range reporting data structures for the plane with

- $O(\sqrt{\log(1/\varepsilon)} + k)$ query time, with $O(\min(1/\varepsilon^2 \log(1/\varepsilon), n \log n))$ storage space and $O(\min(n + 1/\varepsilon^2 \log(1/\varepsilon), n \log n))$ preprocessing time, or
- $O(\log \log(1/\varepsilon) + k)$ query time, with $O(\min(1/\varepsilon^2 \log(1/\varepsilon), n \log n))$ storage space and $O(n + 1/\varepsilon^3 \log(1/\varepsilon))$ preprocessing time.

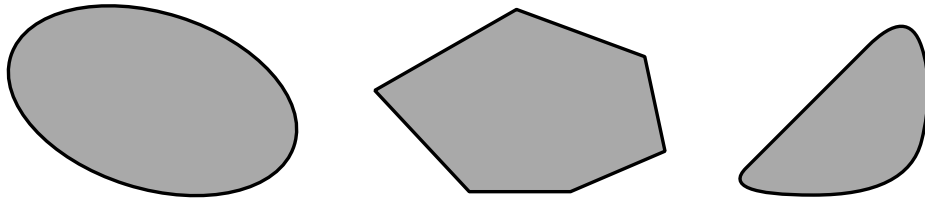


Figure 4.6: Examples of convex ranges.

4.2 Convex Range Searching

In this section we consider \mathcal{R} to be an arbitrary set of convex regions. Some examples of convex regions are shown in Figure 4.6.

As the range space is quite general, we need to assume that some operations on the range can be performed quickly enough. We use the same assumption used in [12], that we call the *unit-cost test assumption*: given $R \in \mathcal{R}$ and a d -dimensional hypercube, in constant time we can determine whether the hypercube is contained within R , is disjoint of R , or neither. We can restrict the unit-cost test assumption only to the case when the d -dimensional hypercube R is a quadtree box (defined in Section 2.2).

There are no known exact range searching data structures for general convex ranges. Agarwal and Matoušek [4] study the range space formed by elementary cells. An *elementary cell* is defined as the conjunction of a constant number of polynomial inequalities of constant degree.

Arya and Mount [12] addressed convex range searching in the relative model, providing a data structure with $O(\log n + 1/\varepsilon^{d-1})$ query time and $O(n)$ space. Their data structure is somewhat similar, but more complicated than the one we describe here. Instead of storing the quadtree boxes in a quadtree,

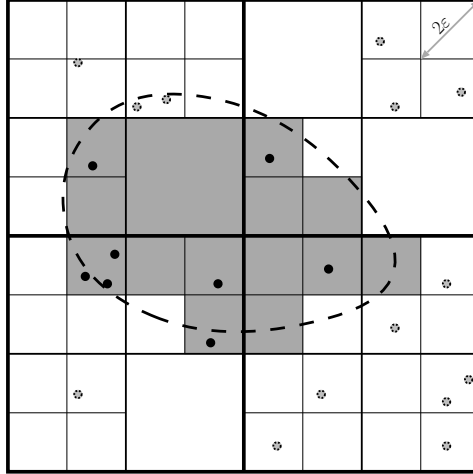


Figure 4.7: Convex range query. The points inside the gray quadtree boxes are counted for a query with the convex range represented by the dashed line.

their data structure uses a BBD-tree (Section 2.2). The depth of the BBD-tree is $O(\log n)$, regardless of the value of ε , which is part of the query, and not of the data structure.

We define \mathcal{G} as the set of quadtree boxes of diameter at least 2ε . Note that $|\mathcal{G}| = O(1/\varepsilon^d)$. The function $g(R)$ is defined as the set of quadtree boxes from \mathcal{G} whose centers are contained in \mathcal{R} . The data structure (\mathcal{G}, g) answers convex range queries in $O(1/\varepsilon^{d-1})$ time. We can efficiently compute $g(R)$ in the real RAM model if we assume that we can determine whether a range intersects a quadtree box in $O(1)$ time (as in [12]). A variation of this data structure uses a compressed quadtree to store arbitrarily small quadtree boxes. In this variation, the parameter ε only needs to be provided at query time, and the storage space is $O(n)$.

To analyze the query time, we consider the recursion tree of the query algorithm. The diameter of the quadtree boxes at level ℓ is $O(2^{-\ell})$. The query

algorithm only makes a recursive call when v_{\square} intersects the boundary of R . It follows from Lemma 2.4 that the number of recursive calls at level ℓ is $O(2^{\ell(d-1)})$. Summing the number of recursive calls for all levels we have

$$\sum_{\ell=1}^{\log(O(1/\varepsilon))} O(2^{\ell(d-1)}),$$

and conclude that the query time is $O(1/\varepsilon^{d-1})$ for $d \geq 2$.

The height of T is $O(\log(1/\varepsilon))$, and insertions can be performed in $O(\log(1/\varepsilon))$ time. To insert a point p , it suffices to perform a recursive search for p , creating two new quadtree boxes, and adding the weight of p to the boxes in the search path.

In the group version of the problem, deletions can also be performed in $O(\log(1/\varepsilon))$ time, without changing the data structure. The delete procedure is similar to the insertion described above.

On the other hand, if a subtraction operation is not available, we can only perform deletions if we explicitly store the elements. Using a tournament tree [15], insertions and deletions can be performed in $O(\log(n/\varepsilon))$ time. This approach requires $O(n)$ space.

If we explicitly store the elements, the storage requirement becomes $O(n)$, but we can build the data structure without knowing ε , and make ε part of the query.

4.3 Range Sketching

The semigroup definition of range searching focuses on queries whose results can be stored in a constant number of memory positions. For example, the result of a counting query is a single number between 0 and n , which is concise, but provides no geometric information about the points in the range. Conversely, a range reporting query provides a complete description of the set of points inside the query range, that can be as large as $\Theta(n)$. In this section, we consider the *range sketching problem*, which tries to fill this gap.

Let P be a set of n points in \mathbb{R}^d and s be a parameter specified at query time. Given a range $R \in \mathcal{R}$, the result of a *range sketching query* $q_s(R)$ is a set of pairwise disjoint hypercubes Q_1, \dots, Q_k , such that, for $1 \leq i \leq k$:

- $Q_i \cap R \neq \emptyset$,
- $s/2 < \text{diam}(Q_i) \leq s$,
- $Q_i \cap P \neq \emptyset$, and
- Q_i is associated with a weight

$$w(Q_i) = \sum_{p \in P \cap Q_i} w(p).$$

An example of range sketching is presented in Figure 4.8. The output size k is proportional to the smallest possible number of non-empty quadtree boxes of diameter at most s that intersect R . Let $\delta > 0$ be an arbitrary constant, and R^+ denote the locus of the points within distance at most $\delta \text{diam}(R)$ from R . Let k' be the smallest number of non-empty quadtree boxes of diameter

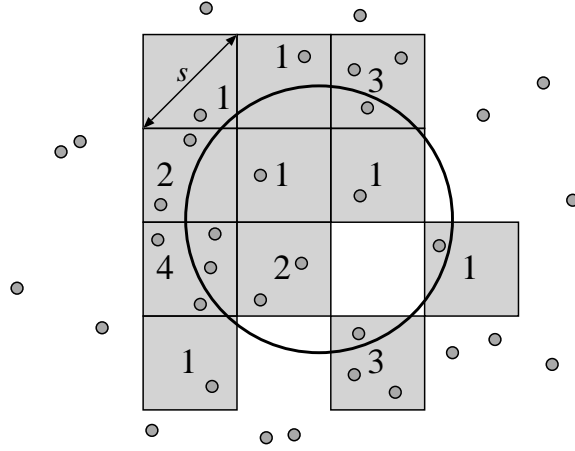


Figure 4.8: Example of range sketching.

at most s that intersect R^+ . In this section, we show how to answer range sketching queries in $O(\log n + k')$ time, with $O(n)$ storage space and $O(n \log n)$ preprocessing time, for arbitrary ranges. We use the same unit-cost test assumption as in Section 4.2. Note that answering a range sketching query requires $\Omega(\log n + k)$ time in any decision tree based model and both k and k' are $O(1 + (\text{diam}(R)/s)^d)$.

Let T be a compressed quadtree for the set P . We build a finger tree T' for T as in Section 2.2. Given a query quadtree box Q , a *cell query* consists of finding the only quadtree box Q' in T such that $P \cap Q = P \cap Q'$. Using T' , a cell query can be answered in $O(\log n)$ time (see Section 2.2).

Let $\delta > 0$ be a constant. Let a be a valid diameter for a quadtree box, with a between $\delta \text{diam}(R)$ and $\delta \text{diam}(R)/2$. To answer a range sketching query, we determine the set A of quadtree boxes of diameter a that intersect R . By Lemma 2.3, we have $|A| = O(1)$. Let B be the set formed by the result of cell queries for each element of A . Note that $|B| = |A| = O(1)$, and B can be computed in $O(\log(n))$ time.

We answer the range sketching query by applying the following query algorithm for each $v \in B$:

1. If $v_{\square} \cap R = \emptyset$, then return.
2. If $|\dot{v}| = 1$, then return v .
3. If $\text{diam}(v_{\square}) \leq s$, then return v .
4. Otherwise, recursively return $\bigcup_c q(R, c)$, for all children c of v .

To analyze the time complexity, we look at the set of recursion trees of the query algorithm above for each node $v \in B$. By construction, the subtrees rooted at the nodes in B are disjoint and only contain points within distance δ from R . Because T is a compressed quadtree, every non-leaf node v has at least 2 children, and contains more than one data point in \dot{v} . Therefore, the number of leaves and also the number of internal nodes in the recursion trees is at most k' . The following theorem summarizes this result.

Theorem 4.3. *There exists a linear space data structure which answers a range sketching query in $O(\log n + k')$ time, where k' is the smallest number of nonempty quadtree boxes of diameter at most s that are within distance δ from the query range, and $\delta > 0$ is an arbitrary constant.*

Chapter 5

Halfspace Range Searching

In the halfspace range searching problem, the range space \mathcal{R} is the set of all halfspaces. Because of the unbounded nature of the ranges, halfspace range searching cannot be solved in the relative model. In this chapter, we present approximate halfspace range searching data structures for the semigroup, idempotent, and emptiness versions of the problem, and also an exact halfspace range searching data structure for the idempotent version.

In Section 5.1, we present previously known upper and lower bounds for exact halfspace range searching. In Section 5.2, we show that the approximate version can be solved in $O(1)$ query time, $O(1/\varepsilon^d)$ space, and $\tilde{O}(n + 1/\varepsilon^d)$ preprocessing time. This is noteworthy, given the high complexity of the exact version. In Section 5.3, we make use of idempotence to achieve a space-time tradeoff, building a data structure with $m \geq 1/\varepsilon^{(d+1)/2}$ storage space and $O(1/m\varepsilon^d)$ query time. In Section 5.5, we use the approximate idempotent data structure to improve the best known bounds of exact range searching, in the semigroup arithmetic model, when the points are uniformly distributed in

the unit cube. In Section 5.4, we improve the idempotent data structure for the case of emptiness queries.

Without loss of generality, we consider the range space \mathcal{R} to be the set of halfspaces of the form $x_d \leq b + a_1x_1 + \dots + a_{d-1}x_{d-1}$ with $-1 \leq a_1, \dots, a_{d-1} \leq 1$. We call the terms a_1, \dots, a_{d-1} *slopes* and b the x_d -*intercept*. An arbitrary halfspace can be converted into this form through an appropriate rotation. As only $2d$ different rotations are necessary, one data structure can be kept for each rotated set of points, without changing our asymptotic results.

5.1 Previous Results

Halfspace ranges are well studied not only because of their simple geometric description, but also because halfspace ranges are related to two other important types of ranges: simplices and spheres. It is believed that the complexity of exact halfspace range searching is very similar to the complexity of exact simplex range searching in both the semigroup and group versions [43]. Actually, most data structures for exact halfspace range searching in the semigroup version can handle simplex ranges with little or no modifications. In the emptiness and reporting versions, though, there are more efficient data structures for halfspace ranges than there are for simplex ranges. Exact spherical range searching in d -dimensional space can be reduced to exact halfspace range searching in $(d+1)$ -dimensional space by projecting the points onto an appropriate $(d+1)$ -dimensional paraboloid [29].

Unlike orthogonal range searching, there are no known exact data structures with near-linear storage and polylogarithmic query time. We briefly

mention some exact range searching results. There are two good surveys that explain the problem in more detail: [2, 43].

There are essentially two types of exact data structures for the problem. In the first type, the query time is logarithmic and the storage space is exponential (near $O(n^d)$). The best data structure of the first type is due to Matoušek [42], and has $O(\log n)$ query time with $O(n^d/\log^d n)$ storage space. Efficient data structure with logarithmic query time are often described in terms of the dual problem of point location (described in Section 3.3.1). A simple and efficient way to solve the point location problem uses Chazelle's hierarchical cuttings [24].

The second type of data structure uses linear or almost linear space, but the query time is exponential in d . The best data structure is due to Matoušek [42], and has $O(n^{1-\frac{1}{d}})$ query time with $O(n)$ storage space and $O(n^{1+\varepsilon})$ preprocessing time, where ε is an arbitrarily small constant. The structure is based on the idea of simplicial partitions, which we describe in Section 2.2.

In [42], Matoušek gives a space-time tradeoff which gives $O(n/m^{\frac{1}{d}})$ query time with m units of storage space. This data structure is believed to have optimal query time for linear space, and almost matches the simplex range searching lower bound. There may be room for improvement by reducing the preprocessing time to $O(n \log n)$, as well as developing simple, more practical structures with similar complexities.

Brönnimann, Chazelle, and Pach [18] present a lower bound result that if $m > n$ units of storage space are allowed, the query time is

$$\Omega \left(\frac{\left(\frac{n}{\log n} \right)^{1 - \frac{d-1}{d(d+1)}}}{m^{\frac{1}{d}}} \right).$$

The lower bound above uses the semigroup arithmetic model, and holds on the expected case when the data points are uniformly distributed in the unit hypercube. The most efficient exact data structure known for the semigroup version is due to Matoušek [42] and has $O(n/m^{\frac{1}{d}})$ query time with m storage space. For small d , the gap between the best general lower bound and the best upper bound is significant. For example, when $m = O(n)$ and $d = 2$, there is a $\tilde{\Omega}(n^{1/3})$ lower bound, and a $O(n^{1/2})$ upper bound.

Arya, Malamatos, and Mount [10] studied the importance of the semigroup being idempotent. A semigroup $(\mathbf{S}, +)$ is *idempotent* if $x + x = x$ for all $x \in \mathbf{S}$, and is *integral* if $kx \neq x$ for all $x \in \mathbf{S} \setminus \{0\}$, and $k \in \mathbb{N}^+$. They showed that, when the semigroup is integral, the lower bound for exact halfspace range searching can be improved to

$$\Omega \left(\frac{n}{m^{\frac{d+1}{d^2+1}} \log n} \right).$$

They also used idempotence to develop more efficient spherical range searching data structures, in the relative model. We show that idempotence can be used not only to improve halfspace range searching in the absolute model, but also in the exact version assuming uniform distribution.

Chazelle, Lin and Magen [26] considered the approximate version of the problem in the absolute model, without assuming that d is a constant. They present a data structure with $\tilde{O}((d/\varepsilon)^2)$ query time and $dn^{O(1/\varepsilon^2)}$ storage.

Their data structure is based on ball range searching in the Hamming cube.

5.2 Approximate Semigroup Version

In this section, we describe a data structure to solve approximate halfspace range searching with $O(1)$ query time, $O(1/\varepsilon^d)$ storage space, and $\tilde{O}(n + 1/\varepsilon^d)$ preprocessing time. The general idea is to define a sufficiently large set \mathcal{G} of halfspaces, so that any query halfspace is approximated by some halfspace in \mathcal{G} . As no two halfspaces in \mathcal{G} are too similar to each other, efficient preprocessing requires building approximate data structures for subdivisions of the unit cube.

We define the set of generators \mathcal{G} as the set that contains \emptyset , $[0, 1]^d$, and the halfspaces whose boundary intersect the unit hypercube and have the slopes and the x_d -intercept as multiples of a parameter ε' to be specified later (Figure 5.1). Therefore, \mathcal{G} contains $O(1/\varepsilon'^d)$ halfspaces. Given a halfspace $R \in \mathcal{R}$, the function $g(R)$ is the set containing the single halfspace obtained by rounding all slopes and the x_d -intercept of R to the closest multiple of ε' . If the boundary of R does not intersect the unit hypercube, then $g(R)$ is defined as $\{[0, 1]^d\}$ if $[0, 1]^d \subseteq R$, and $\{\emptyset\}$ if $R \cap [0, 1]^d = \emptyset$.

Lemma 5.1. *The halfspace in $g(R)$ ($d\varepsilon'/2$)-approximates R .*

Proof. The case when the boundary of R does not intersect the unit hypercube is trivial. Let a_1, \dots, a_{d-1}, b denote the slopes and the x_d -intercept of R , and $a'_1, \dots, a'_{d-1}, b'$ denote the slopes and the x_d intercept of $g(R)$. With some simple manipulations, we can show that the maximum distance δ between the boundaries of R and R' , along the x_d axis and inside the unit box, is

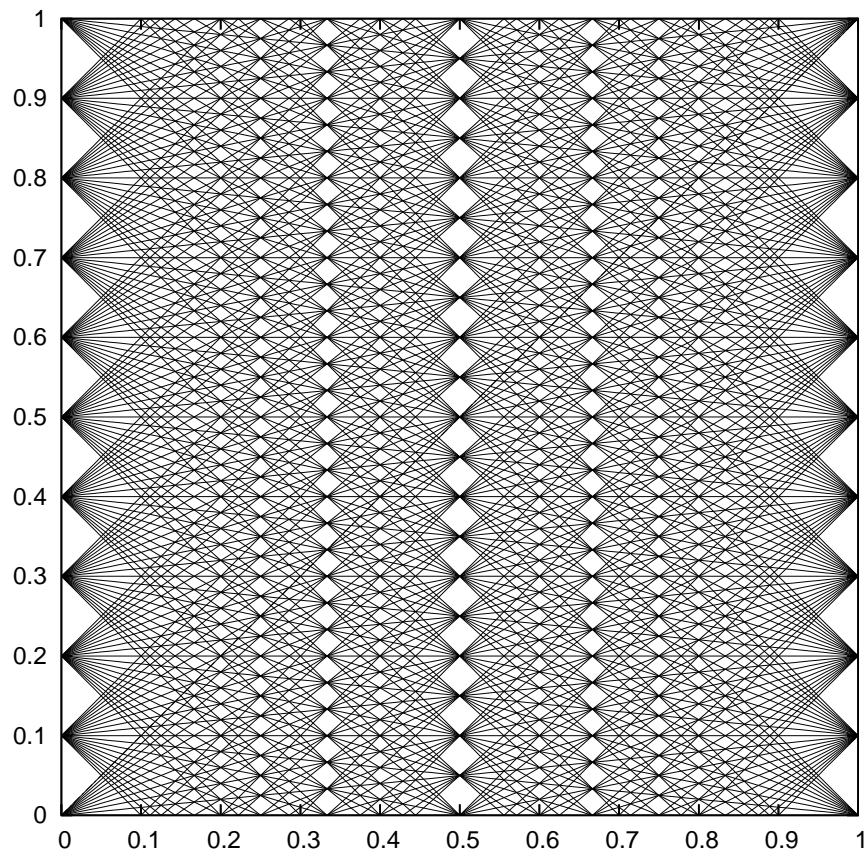


Figure 5.1: Boundaries of the halfspaces in \mathcal{G} with $\varepsilon' = 0.1$.

$$\delta \leq |b - b'| + \sum_{i=1}^{d-1} |a_i - a'_i|.$$

As b' and a'_i are obtained by rounding b and a_i to the closest multiples of ε' , we have $|b - b'| \leq \varepsilon'/2$, and $|a_i - a'_i| \leq \varepsilon'/2$. Therefore, $\delta \leq d\varepsilon'/2$. \square

To build an ε -approximate data structure, we set $\varepsilon' = 2\varepsilon/d$. Using Lemma 5.1, we have:

Theorem 5.2. *(\mathcal{G}, g) is an ε -approximate halfspace range searching data structure with $O(1/\varepsilon^d)$ storage space, $O(1)$ query time, and $\tilde{O}(n + 1/\varepsilon^d)$ preprocessing time.*

It is easy to implement the query algorithm in the real RAM model, without changing the storage space or the query time, as long as integer division is available. Preprocessing the data structure efficiently is not trivial, though. We discuss two ways to perform the preprocessing. Both approaches produce an approximation factor greater than ε , therefore the parameter ε needs to be scaled accordingly. The first way is similar to Chan's discrete Voronoi diagram construction [19]. Consider hyperplanes parallel to one arbitrary orthogonal hyperplane to be horizontal.

1. Like cutting a loaf of bread, partition the unit hypercube in $1/\varepsilon$ horizontal slices, each of thickness ε .
2. Project the points from each slice to a $(d - 1)$ -dimensional horizontal hyperplane that is parallel to the slice boundary and passes through the center of the slice.

3. Recursively compute a $(d - 1)$ -dimensional approximate halfspace range searching data structure for each slice. Use the 0-dimensional case as a trivial base case.
4. For each generator $G \in \mathcal{G}$ ($|\mathcal{G}| = 1/\varepsilon^d$), and each slice s (out of $1/\varepsilon$ slices), perform a query for the intersection of G and s using the data structure for the slice s . Make $w(G)$ the sum of the results of all queries from generator G .

Disregarding the additive term of $O(n)$, the preprocessing time $T(d)$ for a d -dimensional data structure satisfies $T(0) = O(1)$, and $T(d) = O(1/\varepsilon^{d+1}) + (1/\varepsilon)T(d - 1) = O(1/\varepsilon^{d+1})$.

The data structure obtained this way will not be an ε -approximate data structure, as the error accumulates through the d levels of the recursion. Additionally, projecting the points from a slice into a hyperplane adds an error of $\varepsilon/2$ at each level. Nevertheless, the data structure is $O(\varepsilon)$ -approximate (more precisely, $(3d\varepsilon/2)$ -approximate).

The second way to preprocess the data structure is:

1. Divide the unit hypercube in 2^d identical hypercubes.
2. Recursively compute an approximate halfspace range searching data structure for each subdivision. Use the case when the hypercube has diameter ε as a base case.
3. For each generator $G \in \mathcal{G}$ ($|\mathcal{G}| = 1/\varepsilon^d$), and each subdivision s (out of 2^d subdivisions), perform a query for the intersection of G and s using the

data structure for the subdivision s . Make $w(G)$ the sum of the results of all queries from generator G .

Disregarding the additive term of $O(n)$, the preprocessing time $T(\delta)$ for a data structure of diameter δ satisfies $T(\varepsilon) = O(1)$, and $T(\delta) = O(\delta/\varepsilon^d) + 2^d T(\delta/2) = O(\delta \log(\delta/\varepsilon)/\varepsilon^d)$.

We should note that the error accumulates through $O(\log(1/\varepsilon))$ levels. Consequently, the data structure is $O(\varepsilon \log(1/\varepsilon))$ -approximate. We can set $\varepsilon = \varepsilon'/\log(1/\varepsilon')$, and obtain an $O(\varepsilon')$ -approximate data structure in $O(n + \log^{d+1}(1/\varepsilon')/\varepsilon'^d)$ preprocessing time.

5.3 Approximate Idempotent Version

In this section, we make use of idempotence to achieve a space-time tradeoff, building a data structure with $m \geq 1/\varepsilon^{(d+1)/2}$ storage space and $O(1/m\varepsilon^d)$ query time. The idea is to use a set of properly placed large balls as generators (Figure 5.2). There are two sources of approximation error: one comes from the fact that we are approximating flat surfaces with balls, and the second one comes from the fact that we may use balls that are not exactly tangent to the surface being approximated. First, we present a scheme involving an infinite number of generators, which addresses the first issue. Then, we reduce this to a finite set of generators.

Let $r > \sqrt{d} + 1$ be a constant, and let $\varepsilon < 1/2$ be an approximation parameter. We define \mathcal{G}' to be the set of balls B of radius r such that B is centered at (x_1, \dots, x_d) where x_1, \dots, x_{d-1} are multiples of $\sqrt{r\varepsilon}/d$. Note that $|\mathcal{G}'|$ is infinite, and there is no restriction on x_d . Given a halfspace $R \in \mathcal{R}$, let

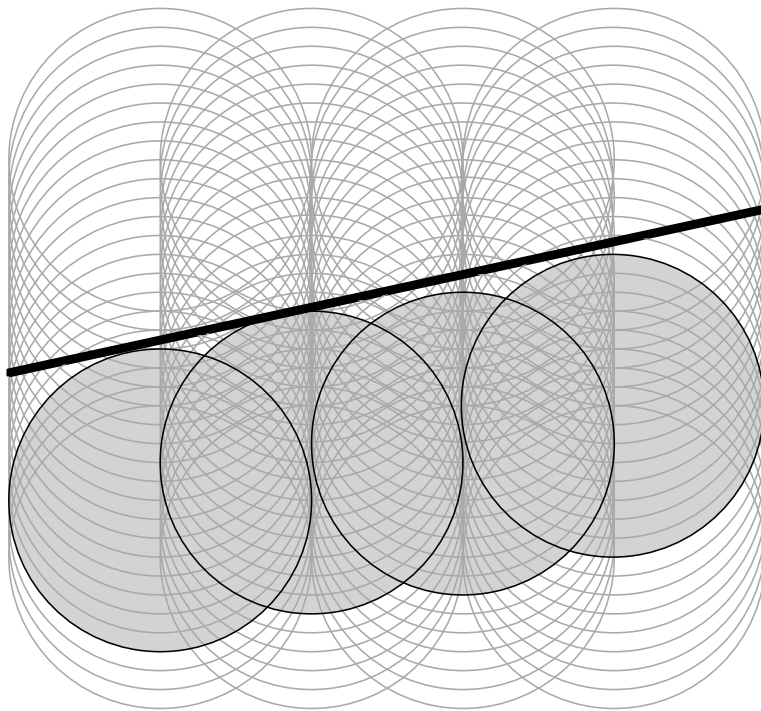


Figure 5.2: General idea of the idempotent version of the halfspace range searching data structure.

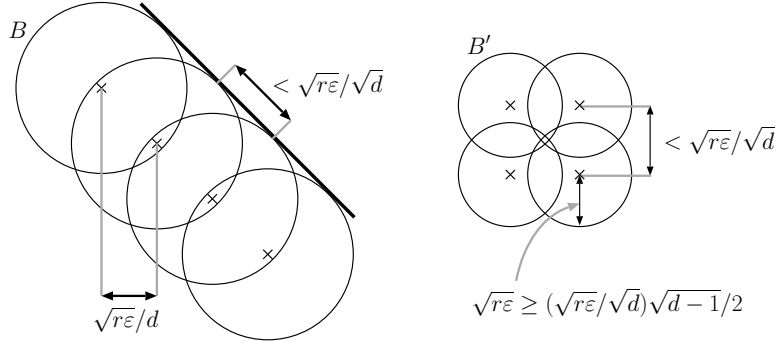


Figure 5.3: Proof of Lemma 5.3.

$g'(R)$ be the subset of balls from \mathcal{G}' that are tangent to the boundary of R and are contained in R .

Lemma 5.3. (\mathcal{G}', g') $(\varepsilon/2)$ -generates \mathcal{R} .

Proof. As $r > \sqrt{d}$, it suffices to show that every point in the boundary of R , and inside the unit hypercube, is within distance at most $\varepsilon/2$ of some ball in $g(R)$.

Look at the point of tangency $t(B)$ between a ball $B \in g(R)$ and the boundary of R . We can prove (using the Pythagorean Theorem) that there is a $(d-1)$ -dimensional ball B' , on the surface of R , of radius greater than $\sqrt{r\varepsilon}$, and centered at $t(B)$, such that all points inside B' are within distance $\varepsilon/2$ of B . The points $t(B)$, for $B \in g(R)$, form a grid of distance at most $\sqrt{r\varepsilon}/\sqrt{d}$ on the surface of R . As the radius of B' is at least $\sqrt{r\varepsilon}$, every point in the surface of R is contained in at least one ball B' (Figure 5.3). \square

We now define $\mathcal{G} \subset \mathcal{G}'$ as the set of balls B such that

1. B has radius r ;

2. there is a hyperplane h with slopes between -1 and 1 , that is tangent to B at point p , with $p \in [-\sqrt{r\varepsilon}/d, 1 + \sqrt{r\varepsilon}/d]^{d-1} \times [0, 1 + \sqrt{d}]$; and
3. B is centered at (x_1, \dots, x_d) where x_1, \dots, x_{d-1} are multiples of $\sqrt{r\varepsilon}/d$, and x_d is a multiple of $\varepsilon/2$.

We define a *column* of \mathcal{G} as the set of balls with centers having the same x_1, \dots, x_{d-1} coordinates. The function $g(R)$ is obtained by replacing each ball B' from $g'(R)$ that approximates the boundary of R within $[0, 1]^d$, with the closest ball $B \in \mathcal{G}$ such that $B \subset R$. Intuitively, the ball B is the ball immediately below B' in the same column.

Theorem 5.4. *(\mathcal{G}, g) is an ε -approximate halfspace range searching data structure, for the idempotent version, with internal approximation, $m \geq 1/\varepsilon^{(d+1)/2}$ storage space, $O(1/m\varepsilon^d)$ query time, and $O(n + \log^{d+1}(1/\varepsilon)/\varepsilon^d)$ preprocessing time.*

Proof. The balls in \mathcal{G} are arranged in $O((r/\varepsilon)^{(d-1)/2})$ columns, and there are $O(1/\varepsilon)$ balls in each column, therefore the storage space is $|\mathcal{G}| = m = O(r^{(d-1)/2}/\varepsilon^{(d+1)/2})$. The query time is $O((1/r\varepsilon)^{(d-1)/2}) = O(1/m\varepsilon^d)$, for $r > 1 + \sqrt{d}$. We defer the discussion on how to preprocess the data structure until Section 6.2.

The set \mathcal{G} has a ball within distance $\varepsilon/2$ from any ball in \mathcal{G}' that can ε -approximate the boundary of a halfspace in \mathcal{R} inside the unit hypercube. From Lemma 5.3, we conclude that the set $g(R)$ ε -approximates R . \square

5.4 Approximate Emptiness Version

In this section, we show how to reduce the storage space of the idempotent data structure for the special case of range emptiness queries, that is, for the semigroup $(\{0, 1\}, \vee)$. In the plane, the exact problem can be solved in $O(\log n)$ query time and $O(n)$ space using a binary search in the convex hull [51]. For $d = 3$, we can also have logarithmic query time with almost linear $O(n \log n)$ space [30]. For $d > 3$, the best known upper bounds are only slightly better than the semigroup version. Our approach is to modify the data structure from Section 5.3 to obtain some kind of monotonicity, and then apply compression to reduce the storage space.

For a set of points B , let $\tau(B) = \{(x_1, \dots, x_d) : \exists(x_1, \dots, x_{d-1}, x'_d) \in B \text{ with } x'_d > x_d\}$. If B is a ball, then $\tau(B)$ is a bullet-shaped object formed by the ball and a semifinite cylinder extending downwards. We modify the set \mathcal{G} from Section 5.3 into a set $\mathcal{G}_{emp} = \{\tau(B) : B \in \mathcal{G}\}$. It is not hard to see that Theorem 5.4 still holds if we replace \mathcal{G} with \mathcal{G}_{emp} .

The set \mathcal{G}_{emp} can be compressed, because generators $G \in \mathcal{G}_{emp}$ with the same x_1, \dots, x_{d-1} coordinates can be ordered by their x_d coordinates, and the value of $w(G)$ can only change once. Therefore, the storage space requirement is reduced by a factor of $O(1/\varepsilon)$. Preprocessing can be performed using the technique described in Section 6.2, but using a halfbox quadtree for the emptiness version constructed using an approach similar to Chan's discrete Voronoi diagram construction [19].

Theorem 5.5. *There is an ε -approximate halfspace range searching data structure for the emptiness version with $m \geq 1/\varepsilon^{(d-1)/2}$ storage space,*

$O(1/m\varepsilon^{d-1})$ query time, and $O(n + 1/\varepsilon^d)$ preprocessing time.

Next, we describe another approach to this problem, which consists of computing an ε -kernel [3, 19] containing $O(1/\varepsilon^{\frac{d-1}{2}})$ points, and then using an exact halfspace emptiness data structure with the ε -kernel as the set of points. The latter approach attains lower query times for the case of $O(1/\varepsilon^{(d-1)/2})$ space, but involves complex data structures from [41] for $d > 3$.

An important tool in approximating geometric measures is called ε -kernel, and its applications are explained in detail in [3]. Let $\omega(u, P)$ denote the width of P with respect to direction u . An ε -kernel is defined as a subset P' of P such that

$$(1 - \varepsilon)\omega(u, P) \leq \omega(u, P'),$$

for all directions $u \in \mathbb{R}^d$.

We define an ε -hull of P as a subset $P' \subseteq P$ such that, for all directions $u \in \mathbb{R}^d$,

$$\omega(u, P) \leq \omega(u, P') + \varepsilon.$$

As $P \subset [0, 1]^d$, an ε -kernel is a $(\sqrt{d}\varepsilon)$ -hull. Chan showed that an ε -kernel of size $O(1/\varepsilon^{(d-1)/2})$ can be constructed in $O(n + 1/\varepsilon^{d-(3/2)})$ time [19]. We show that we can use the point approximation technique, using P' as the ε -hull.

Lemma 5.6. *Let P be a set of data points, P' an ε -hull of P , and h a halfspace with boundary ∂h . If $P' \cap h$ is empty, then $P \cap h$ contains only points within distance at most ε from ∂h .*

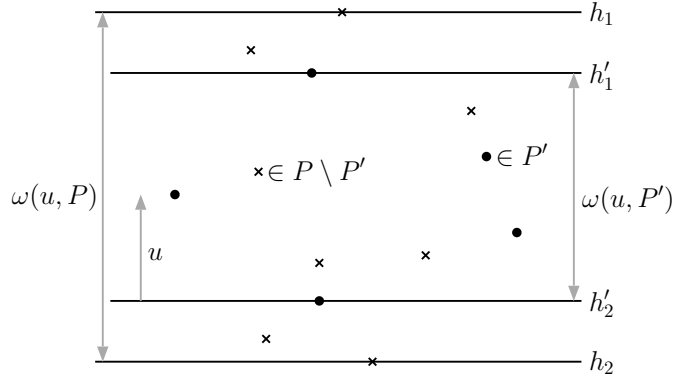


Figure 5.4: Proof of Lemma 5.6.

Proof. Let h_1, h_2 (respectively h'_1, h'_2) denote the two supporting hyperplanes for P (resp. P') that are parallel to ∂h , and perpendicular to the direction u (Figure 5.4). If $\partial h \cap h'_1 \neq \emptyset$ or $\partial h \cap h'_2 \neq \emptyset$, the theorem holds because $P' \cap h \neq \emptyset$. Without loss of generality we say h'_1 separates ∂h from h'_2 . If $\partial h \cap h_1 = \emptyset$, the theorem holds because $P \cap h = \emptyset$. We only need to handle the case where h contains h_1 but not h'_1 . As the distance between h_1 and h'_1 is at most ε because of the ε -hull definition, the theorem holds. \square

As $P' \subseteq P$, we know that if $P' \cap h \neq \emptyset$, then $P \cap h \neq \emptyset$. Using this and Lemma 5.6, we conclude that an exact query with P' is an ε -approximate query for P . Some of the best known near-linear space data structures to answer exact halfspace emptiness queries are:

d	Query Time	Size	Reference
$d = 2$	$O(\log n)$	$O(n)$	Preparata and Shamos [51]
$d = 3$	$O(\log n)$	$O(n \log n)$	Dobkin et al. [30]
$d \geq 4$	$O(n^{1-1/\lfloor d/2 \rfloor} 2^{O(\log^* n)})$	$O(n)$	Matoušek [41]
$d \geq 4$	$O(\log n)$	$\tilde{O}(n^{\lfloor d/2 \rfloor})$	Matoušek and Schwarzkopf [46]

Replacing n with $1/\varepsilon^{(d-1)/2}$ we have:

d	Query Time	Size
$d = 2$	$O(\log(1/\varepsilon))$	$O(1/\varepsilon^{(d-1)/2})$
$d = 3$	$O(\log(1/\varepsilon))$	$O(1/\varepsilon^{(d-1)/2} \log(1/\varepsilon))$
even $d \geq 4$	$O(1/\varepsilon^{(d-3)/2+1/d} 2^{O(\log^*(1/\varepsilon))})$	$O(1/\varepsilon^{(d-1)/2})$
odd $d \geq 5$	$O(1/\varepsilon^{(d-3)/2}) 2^{O(\log^*(1/\varepsilon))}$	$O(1/\varepsilon^{(d-1)/2})$
even $d \geq 4$	$O(\log(1/\varepsilon))$	$\tilde{O}(1/\varepsilon^{d(d-1)/4})$
odd $d \geq 5$	$O(\log(1/\varepsilon))$	$\tilde{O}(1/\varepsilon^{(d-1)^2/4})$

The data structures described in the last two lines of the table above are not as efficient as the much simpler data structure with $O(1)$ query time and $O(1/\varepsilon^{(d-1)})$ space from Theorem 5.5. The following theorem summarizes the results in this section:

Theorem 5.7. *There is an ε -approximate halfspace range searching data structure for the emptiness version with*

- $O(\sqrt{1/\varepsilon})$ space, and $O(\log(1/\varepsilon))$ query time, for $d = 2$;
- $O(\sqrt{1/\varepsilon} \log(1/\varepsilon))$ space, and $O(\log(1/\varepsilon))$ query time, for $d = 3$;
- $O(1/\varepsilon^{(d-1)/2})$ space, and $O(1/\varepsilon^{(d-3)/2+1/d} 2^{O(\log^*(1/\varepsilon))})$ query time, for even $d \geq 4$;
- $O(1/\varepsilon^{(d-1)/2})$ space, and $O(1/\varepsilon^{(d-3)/2} 2^{O(\log^*(1/\varepsilon))})$ query time, for odd $d \geq 5$.

5.5 Exact Idempotent Version

In this section, we show how to use the approximate idempotent data structure to build an exact halfspace range searching data structure. The data structure makes use of idempotence to improve over the most efficient exact data structure previously known. This exact data structure is defined in the semigroup arithmetic model [10, 18, 22], and has $O(n^{1-2/(d+1)})$ expected query time with $O(n)$ space, matching the lower bound proved in [18] up to logarithmic factors. The theoretical importance of the data structure relies on the fact that uniform distribution and the semigroup arithmetic model are also assumed in the lower bound proved in [18]. Therefore, we open some important theoretical and practical questions: Is the average case complexity for uniformly distributed data strictly lower than the worst case complexity? Does the semigroup arithmetic model allow more efficient idempotent halfspace range searching data structures than the real RAM model? We do not provide an efficient way to determine the set of generators used to answer a given query. Consequently, the results hold only for the semigroup arithmetic model. An improved data structure that worked in the real RAM model would be of practical interest, even if it relied on the uniform distribution of the points.

The general idea of the data structure is to properly set the parameter ε used in the data structure from Section 5.3, in order to make the expected number of points in the fuzzy boundary equal to the query time of the approximate data structure. Generators for individual points can be used to count the points in the fuzzy boundary.

In this section, we assume that the n data points are uniformly distributed

in the unit hypercube $[0, 1]^d$. Let $m \geq n$ denote the storage space, and $\varepsilon = (nm)^{-1/(d+1)}$. We apply Theorem 5.4 to build an ε -approximate data structure (\mathcal{G}_1, g_1) with $O(m)$ storage space and query time

$$O\left(\frac{1}{m\varepsilon^d}\right) = O\left(\frac{n^{1-1/(d+1)}}{m^{1/(d+1)}}\right) = O(n\varepsilon).$$

As the data structure (G_1, g_1) provides internal approximation, no data points outside R are counted, but some data points inside R and within distance ε from the boundary may not be counted. To answer the query exactly, we set $\mathcal{G}_2 = \{\{p\} : p \in P\}$ and define $g_2(R)$ as the set of generators $\{p\} \in \mathcal{G}_2$ such that $p \in R$ and p is within distance ε from the boundary of R . As the data points are uniformly distributed within the unit hypercube, for any fixed $R \in \mathcal{R}$, the expected number of generators in $g_2(R)$ is $E(|g_2(R)|) = O(n\varepsilon)$.

Let $\mathcal{G}' = \mathcal{G}_1 \cup \mathcal{G}_2$, and $g'(R) = g_1(R) \cup g_2(R)$. To use the standard semigroup arithmetic model definition of generators, let \mathcal{G} denote the set of linear forms $\sum_{p \in P \cap G'} w(p)$ for all $G' \in \mathcal{G}'$. In summary, we have the following.

Theorem 5.8. *\mathcal{G} is an exact halfspace range searching data structure with $m \geq n$ storage space and $O(n^{1-1/(d+1)}/m^{1/(d+1)})$ expected query time, in the semigroup arithmetic model and assuming that the points are uniformly distributed in a hypercube. The query time is $O(n^{1-2/(d+1)})$ when $m = n$.*

Theorem 5.8 matches the lower bound of $\tilde{\Omega}(n^{1-(d-1)/d(d+1)}/m^{\frac{1}{d}})$ [18] for the case of $m = n$ (up to logarithmic factors). The lower bound is also in the semigroup arithmetic model, and also assumes that the points are uniformly distributed inside the unit hypercube. Therefore, improving the lower bound

for the case of $m = n$, if at all possible, would require either a different model of computation or a different set of data points.

Chapter 6

Halfbox Quadtree

In this chapter, we introduce a data structure called the *halfbox quadtree*, and present some applications. In Section 6.1, we define the data structure, and explain how to perform the preprocessing. In Section 6.2, we analyze the query time of the halfbox quadtree for spherical ranges. We also show how an additional set of generators can be used to provide a space-time tradeoff. In Section 6.3, we show how to use the halfbox quadtree to answer approximate nearest neighbor queries. In Section 6.4, we analyze the query time of the halfbox quadtree for simplex ranges, in the group version, and fat simplex ranges, in the semigroup version. In Section 6.5, we modify the halfbox quadtree to be efficient for range reporting. In Section 6.5, we modify the halfbox quadtree to be efficient in the data stream model.

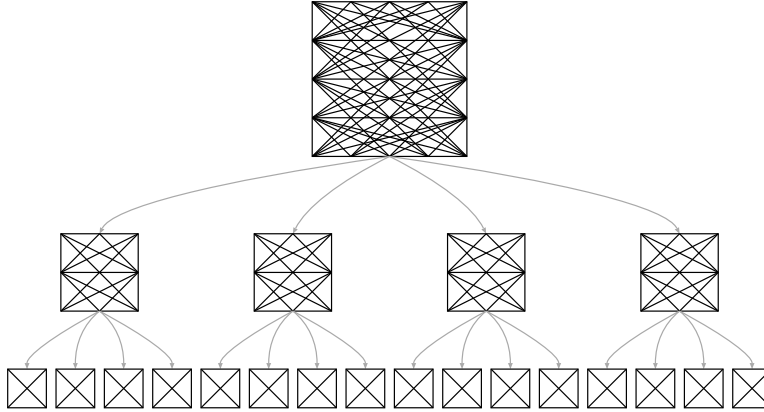


Figure 6.1: Representation of a halfbox quadtree.

6.1 Definition and Preprocessing

A *quadtree box* is a box that can be obtained by recursively dividing the unit hypercube into 2^d identical hypercubes. A *quadtree* is a hierarchical organization of a set of quadtree boxes, where the root of the tree corresponds to the unit hypercube, and the 2^d children of an internal node correspond to the subdivisions of the parent node.

We define a *half quadtree box* (*halfbox* for short) as the intersection of a quadtree box and a halfspace. The *size* of a halfbox is the diameter of the corresponding quadtree box. A *halfbox quadtree* is a quadtree T containing all quadtree boxes of diameter at least ε , where each quadtree box Q is associated with an $(\varepsilon/2)$ -approximate halfspace range searching data structure for the bounding box Q . If we use the semigroup halfspace data structure (from Theorem 5.2), the resulting halfbox quadtree has $O(\log(1/\varepsilon)/\varepsilon^d)$ storage space, and $O(1)$ query time for approximate halfbox range queries. A halfbox quadtree is represented in Figure 6.1.

If we build a semigroup halfspace range searching data structure for each

halfbox from the scratch, the total preprocessing time for the halfbox quadtree is $O(n + \log^{d+2}(1/\varepsilon)/\varepsilon^d)$. We can slightly improve the preprocessing time to $O(n + \log^{d+1}(1/\varepsilon)/\varepsilon^d)$ by noting that the preprocessing of the semigroup halfspace range searching data structure for the unit hypercube already builds a data structure for every quadtree box, in a bottom-up fashion. The fact that the value of ε needs to be modified in the preprocessing algorithm, because of the error accumulation, is easily remedied by discarding unnecessary entries from the data structures at each level.

Theorem 6.1. *The halfbox quadtree uses $O(\log(1/\varepsilon)/\varepsilon^d)$ storage space, can be constructed in $O(n + \log^{d+1}(1/\varepsilon)/\varepsilon^d)$ preprocessing time, and answers ε -approximate halfbox range queries in $O(1)$ time, in the semigroup version.*

6.2 Spherical Range Searching

In spherical range searching, the ranges are Euclidean balls. The exact version of the problem can be reduced to halfspace range searching by projecting the points onto an appropriate $(d + 1)$ -dimensional paraboloid [29]. Spherical range queries in P are equivalent to halfspace range queries in P' . Arya, Malamatos and Mount [8, 10] present approximate data structures, in the relative model, with $m \geq n \log(1/\varepsilon)$ space and $\tilde{O}(n^{1-\frac{1}{d}}/m^{\frac{1}{d}}\varepsilon^{d-1})$ query time, for general semigroups, and $\tilde{O}(n^{1/2-1/2d}/m^{1/2d}\varepsilon^{(d-1)/2})$ query time for idempotent semigroups.

In this section, we show that the halfbox quadtree answers spherical range queries in $O(1/\varepsilon^{\frac{d-1}{2}})$ time. The data structure works for general semigroups. We also show how to reduce the spherical range query time by increasing

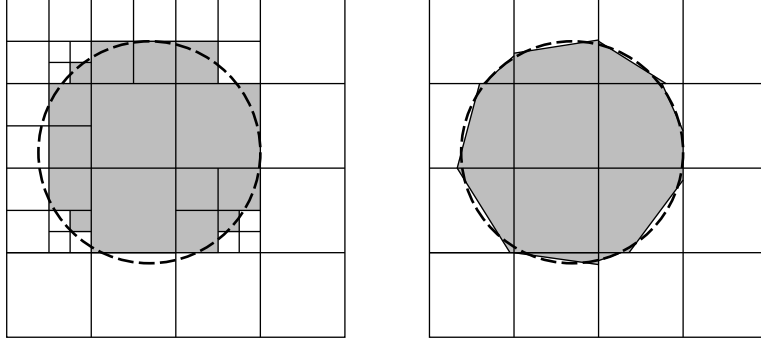


Figure 6.2: Approximating a ball using quadtree boxes (left) and half quadtree boxes (right).

the space, adding some extra generators to the data structure. Figure 6.2 illustrates how a small number of halfboxes can approximate a ball significantly better than a larger number of quadtree boxes.

Theorem 6.2. *The halfbox quadtree is an ε -approximate range searching data structure for spherical ranges with $O(\log(1/\varepsilon)/\varepsilon^d)$ storage space, $O(1/\varepsilon^{(d-1)/2})$ query time, and $O(n + \log^{d+1}(1/\varepsilon)/\varepsilon^d)$ preprocessing time. If the query ball B has radius r , then the query time is $O((\min(r, 1)/(r + 1)\varepsilon)^{(d-1)/2})$.*

Proof. If we start with Q as the unit hypercube, and recursively subdivide Q until we can approximate B by a halfbox associated with Q , we notice that a recursive call is only performed when Q intersects the boundary of B . Let $\delta = O(1/2^i)$ denote the diameter of Q at level i in the recursion tree, and $\Delta = O(\min(r, 1))$ be the diameter of the intersection of B and the unit box. Using Lemmas 2.2, and 2.4, and summing the number of recursive calls, we conclude that the query time is

$$\sum_{i=0}^{\log O(1/\sqrt{r\varepsilon})} (\min(r, 1)/O(1/2^i))^{d-1} = O((\min(r, 1)/(r + 1)\varepsilon)^{(d-1)/2}).$$

□

The halfbox quadtree can be used to preprocess the approximate idempotent halfspace range searching data structure from Theorem 5.4. The total preprocessing time for a data structure of size $m = r^{(d-1)/2}/\varepsilon^{(d+1)/2}$, with $1 < r < 1/\varepsilon$, is

$$O\left(n + \frac{\log^{d+1}(1/\varepsilon)}{\varepsilon^d} + m \left(\frac{1}{r\varepsilon}\right)^{\frac{d-1}{2}}\right) = O\left(n + \frac{\log^{d+1}(1/\varepsilon)}{\varepsilon^d}\right) = \tilde{O}\left(n + \frac{1}{\varepsilon^d}\right).$$

The spherical range searching data structure from Theorem 6.2 has minimum storage space, except for a logarithmic factor (because we could place $1/\varepsilon^d$ points in a grid, and retrieve each individual weight using ranges of radius ε). It is natural to ask how we could improve the query time by increasing the storage space. Let $r > 1$ be a parameter, and consider the set of generators \mathcal{G} formed by \emptyset , $[0, 1]^d$, and the balls B such that:

1. B has radius at most r ,
2. the radius of B is a multiple of $\varepsilon/2$,
3. the boundary of B intersects $[0, 1]^d$, and
4. B is centered at (x_1, \dots, x_d) where x_1, \dots, x_d are multiples of $\varepsilon/2\sqrt{d}$.

Theorem 6.3. *The halfbox quadtree, together with (\mathcal{G}, g) , is an ε -approximate range searching data structure for spherical ranges with $m = r^d/\varepsilon^{d+1} > 1/\varepsilon^{d+1}$ storage space, $O(1/m^{\frac{1}{2}-\frac{1}{2d}}\varepsilon^{d-\frac{1}{2}-\frac{1}{2d}})$ query time, and $O(n + m/\varepsilon^{(d-1)/2})$ preprocessing time.*

Proof. The storage space is dominated by $|\mathcal{G}| = O(r^d/\varepsilon^{d+1}) = m$. Any ball $R \in \mathcal{R}$ of radius at most $r > 1$ is ε -approximated by a ball $g(R) \in \mathcal{G}$. Therefore, queries with radius at most r can be answered in $O(1)$ time. Using the halfbox quadtree, queries with radius greater than r can be answered in

$$O\left(\left(\frac{1}{r\varepsilon}\right)^{\frac{d-1}{2}}\right) = O\left(\frac{1}{m^{\frac{1}{2}-\frac{1}{2d}}\varepsilon^{d-\frac{1}{2}-\frac{1}{2d}}}\right) \text{ time.}$$

□

In the low storage version, Theorem 6.3 gives a data structure with $O(1/\varepsilon^{d+1})$ storage space, $O(1/\varepsilon^{(d-1)/2})$ query time, and $O(n + 1/\varepsilon^{3d/2+1/2})$ preprocessing time, which is inferior to the data structure from Theorem 6.2. In the high storage version, Theorem 6.3 gives a data structure with $O(1/\varepsilon^{2d+1})$ storage space, $O(1)$ query time, and $O(n + 1/\varepsilon^{5d/2+1/2})$ preprocessing time.

6.3 Approximate Nearest Neighbor

The nearest neighbor problem is a well studied problem in computational geometry. The *nearest neighbor problem* consists of preprocessing a set P of n data points in d -dimensional space in a way that, given a query point q (generally not in P), we can efficiently find the point $p^* \in P$ that minimizes $\text{dist}(q, p^*)$. The point p^* is called the *nearest neighbor (NN)* of the point q . In the planar case, the problem is often solved using the Voronoi diagram and a point location data structure [29]. For $d \geq 3$, no known data structure achieves both near-linear space and polylogarithmic query time.

In the relative model, the *approximate nearest neighbor problem* consists

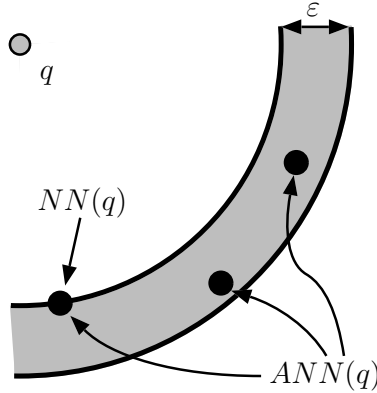


Figure 6.3: Approximate nearest neighbor of a point q .

of finding a point $p' \in P$ such that $\text{dist}(q, p') \leq (1 + \varepsilon)\text{dist}(q, p^*)$. The point p' is called an *approximate nearest neighbor* (*ANN*) of the point q . The literature on approximate nearest neighbor in the relative model is extensive (some references are [6, 7, 11, 13, 36]). Some of the best space time tradeoffs for the problem are presented in [7], and include $O(\log(n/\varepsilon))$ query time with $O(n/\varepsilon^{d-1})$ space and $O(\log n + 1/\varepsilon^{(d-1)/2})$ query time with $O(n)$ space.

In the absolute model, the *approximate nearest neighbor problem* consists of finding a point $p' \in P$ such that $\text{dist}(q, p') \leq \varepsilon + \text{dist}(q, p^*)$ (Figure 6.3). The point p' is called an ε -*nearest neighbor* of the point q . We show how to use the halfbox quadtree (with the approximation error of the halfbox quadtree set to $\varepsilon/2$) to solve the approximate nearest neighbor problem in the absolute model. As usual, we assume that the data points are inside the unit hypercube $[0, 1]^d$, but we do not make any assumptions about the query point q . Note that, if we assume that $q \in [0, 1]^d$, then it is easy to answer approximate nearest neighbor queries in $O(1)$ time with $O(1/\varepsilon^d)$ storage space.

In Section 5.4, we describe a data structure which answers ε -approximate

halfspace emptiness queries in $O(1)$ time, with $O(1/\varepsilon^{d-1})$ storage space and preprocessing time. We can build a half-box quadtree using this data structure and use the half-box quadtree to solve the emptiness version of approximate spherical range searching. The storage space of the halfspace data structure associated with a halfbox quadtree node at level ℓ is $O((1/2^\ell\varepsilon)^{d-1})$. As there are $2^{d\ell}$ nodes at level ℓ , the total size of the half-box quadtree for the emptiness version is

$$\sum_{\ell=0}^{\log(O(1/\varepsilon))} 2^{d\ell} O\left(\left(\frac{1}{2^\ell\varepsilon}\right)^{d-1}\right) = \sum_{\ell=0}^{\log(O(1/\varepsilon))} O\left(\frac{2^\ell}{\varepsilon^{d-1}}\right) = O(1/\varepsilon^d).$$

Consequently, we have:

Theorem 6.4. *There exists a constructive ε -approximate range searching data structure for spherical ranges with $O(1/\varepsilon^d)$ size, $O(n + 1/\varepsilon^d)$ preprocessing time, and $O((1/\varepsilon)^{(d-1)/2})$ query time, for the emptiness version of the problem.*

We can easily modify the data structure to report one of the points inside the approximate spherical range, in case the range is not empty. This type of query is called *one-reporting query*.

Given a query point q , we can determine the minimum and maximum distance between q and the region $[0, 1]^d$ in $O(1)$ time. We call these two distances r_0 and r_1 , respectively. Assuming that $|P| \geq 1$, for otherwise the problem is trivial, we know that $r_0 \leq \text{dist}(q, p^*) \leq r_1$ and also that $r_1 - r_0 = O(1)$.

We denote by $B(c, r)$ the sphere with radius r centered at the point c . We can perform a binary search to find the minimum value of r , with $r_0 \leq r \leq r_1$ and r being a multiple of $\varepsilon/2$, such that $q_{\varepsilon/2}(B(q, r))$ is not empty.

The binary search performs $O(\log(1/\epsilon))$ queries, and the total approximate nearest neighbor query time is $O(1/\epsilon^{(d-1)/2} \log(1/\epsilon))$. The following theorem summarizes the properties of this data structure.

Theorem 6.5. *The one-reporting version of the halfbox quadtree can answer approximate nearest neighbor queries (in the absolute model) with $O(1/\epsilon^d)$ storage space, in $O((1/\epsilon)^{(d-1)/2} \log(1/\epsilon))$ query time, and $O(n + 1/\epsilon^d)$ preprocessing time.*

6.4 Simplex Range Searching

In simplex range searching, the ranges are d -dimensional simplices. If m units of storage are allowed, then the query time is $\Omega(n/\sqrt{m})$ in the plane, and $\Omega((n/\log n)/m^{\frac{1}{d}})$ in d -dimensional space [22]. In the exact version, the most efficient linear size data structure is due to Matoušek [42] and matches the lower bounds up to logarithmic factors, achieving $O(n^{1-\frac{1}{d}})$ query time for the case of linear storage space.

We show how to answer ϵ -approximate simplex queries in $O(\log(1/\epsilon))$ time for $d = 2$ and $O(1/\epsilon^{d-2})$ time for $d \geq 3$, using the halfbox quadtree. Our query algorithm requires the use of a subtraction operation, and so applies only in the group setting. If the query simplex is fat, the use of subtraction is unnecessary, and the same query time is attained in the semigroup version.

We recursively answer a query $q(Q, R)$ with a simplex range R in the quadtree box Q , starting with Q as the unit hypercube.

1. If $Q \cap R = \emptyset$, then we return 0.

2. If $Q \cap R = Q$, then we return the precomputed $w(Q) = \sum_{p \in P \cap Q} w(p)$.
3. If the diameter of Q is less than ε , then we verify whether R contains the center of Q and answer accordingly.
4. If Q does not contain any $(d - 2)$ -faces of R , then there is a set H of at most $d + 1$ halfspaces that form the complement of R , and the halfspaces in H are pairwise disjoint. Then, we can return $w(Q) - \sum_{h \in H} q_\varepsilon(h \cap Q)$, where $h \cap Q$ is a halfbox, and $q_\varepsilon(\cdot)$ is the result of the approximate query using the halfbox quadtree.
5. Otherwise, we return the sum of $q(Q', R)$ for all 2^d subdivisions Q' of Q .

To analyze the query time, we look at the recursion tree of the query algorithm. The diameter of the quadtree boxes at level ℓ is $\Theta(2^{-\ell})$. The query algorithm only makes a recursive call when Q intersects the a $(d - 2)$ -face of R . As R is the intersection of a constant number of halfspaces, the number of $(d - 2)$ -faces of R is also constant. As R is convex, each $(d - 2)$ -face of R inside the unit box is a convex polytope of constant diameter. It follows from Lemma 2.5 that the number of recursive calls at level ℓ is $\Theta(2^{\ell(d-2)})$. Summing the number of recursive calls for all $\log(\Theta(1/\varepsilon))$ levels we conclude:

Theorem 6.6. *The halfbox quadtree is an ε -approximate range searching data structure for simplex ranges, in the group version, with $O(1/\varepsilon^d \log(1/\varepsilon))$ storage space, $O(\log(1/\varepsilon))$ query time for $d = 2$, $O(1/\varepsilon^{d-2})$ query time for $d \geq 3$, and $O(n + \log^{d+1}(1/\varepsilon)/\varepsilon^d)$ preprocessing time.*

Next, we consider the semigroup version of the data structure, where a subtraction operation is not available. In the semigroup version, we require

the query simplex to be fat. A shape R is said to be *fat* when there is some constant α such that, for all d -dimensional balls B not fully containing R , we have

$$\alpha \text{ volume}(R \cap B) \geq \text{volume}(B).$$

Our results only require a weaker definition of a *fat simplex*: a simplex is fat when the angle between any two $(d - 1)$ -dimensional hyperplanes that form the boundary of the simplex is at least a constant. A simplex is *skinny* when it is not fat.

The need for subtraction when answering simplex range queries arises from the fact that a simplex may be skinny (see Figure 6.4 for a 2-dimensional example). In particular, a skinny simplex may have $\Theta(\varepsilon)$ width in a given direction u and $\Theta(1)$ width in any direction perpendicular to u . The d -dimensional volume of such simplex is $O(\varepsilon)$, and it does not fully contain any halfbox of volume greater than $\Theta(\varepsilon^d)$. Therefore, if no subtraction is allowed, $O(1/\varepsilon^{d-1})$ halfboxes are necessary to approximate such skinny simplex.

We recursively answer a query $q(Q, R)$ with a fat simplex range R in the quadtree box Q , starting with Q as the unit hypercube.

1. If $Q \cap R = \emptyset$, then we return 0.
2. If $Q \cap R = B$, then we return the precomputed $w(Q) = \sum_{p \in P \cap Q} w(p)$.
3. If the diameter of Q is less than ε , then we verify whether R contains the center of Q and answer accordingly.
4. If Q contains only one $(d - 1)$ -face h of R , then we return the result of the query halfbox $Q \cap h$.

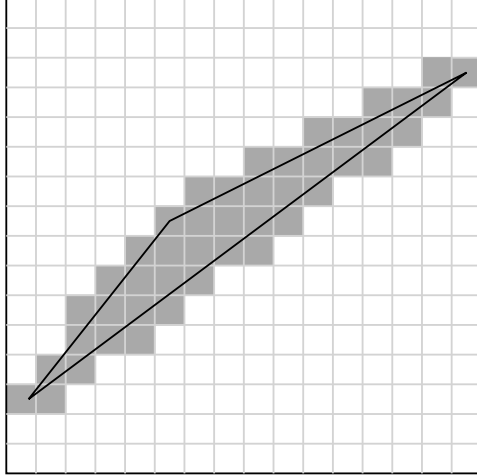


Figure 6.4: Representation of a skinny triangle.

5. Otherwise, we return the sum of $q(Q', R)$ for all 2^d subdivisions Q' of Q .

To analyze the query time, we look at the recursion tree of the query algorithm. The diameter of the quadtree boxes at level ℓ is $\Theta(2^{-\ell})$. The query algorithm only makes a recursive call when Q intersects more than one $(d-1)$ -faces of R . Using the packing Lemma 2.6, we conclude:

Theorem 6.7. *The halfbox quadtree is an ε -approximate range searching data structure for fat simplex ranges, in the semigroup version, with $O(1/\varepsilon^d \log(1/\varepsilon))$ storage space, $O(\log(1/\varepsilon))$ query time for $d = 2$, $O(1/\varepsilon^{d-2})$ query time for $d \geq 3$, and $O(n + \log^{d+1}(1/\varepsilon)/\varepsilon^d)$ preprocessing time.*

Proof. We sum the number of recursive calls level by level in the recursion tree. Using Lemma 2.6 the query time is:

$$\sum_{i=0}^{\log O(1/\varepsilon)} O\left((2^i)^{d-2}\right) = O\left(\left(\frac{1}{\varepsilon}\right)^{d-2} + \log(1/\varepsilon)\right).$$

□

6.5 Approximate Range Reporting

The *range reporting problem* involves preprocessing a set P of n points in \mathbb{R}^d so that given a region R , that is drawn from a predefined set of shapes \mathcal{R} , the set $P \cap R$ can be computed efficiently. The range reporting problem is a special case of the range searching problem where the semigroup is $(2^P, \cup)$, and $w(p) = \{p\}$.

Range reporting deserves special attention because some assumptions made for the semigroup version of the problem do not hold for range reporting in the real RAM. First, storing a generator G with $|G|$ data points requires $\Omega(|G|)$ space (compared to $O(1)$ space for the semigroup version). Second, reporting k points takes $\Omega(k)$ time, which means we can spend more time answering queries that return more points.

The generators in the halfspace range searching data structures from Chapter 5 are intentionally large, so queries can be answered efficiently. The semigroup halfspace range searching data structure has $\Theta(1/\varepsilon^d)$ generators containing $\Theta(1/\varepsilon^d)$ points (assuming the data points are initially approximated to a grid, otherwise the number of points can be even higher). Therefore, an unmodified reporting version of the halfbox quadtree would take $\Theta(1/\varepsilon^{2d})$ storage space. Fortunately, we can exploit some properties of the data structure to reduce the storage space to $O(\log^{d+1}(1/\varepsilon)/\varepsilon^d)$.

Consider an ε -approximate halfbox quadtree T_1 for the unit hypercube. For each halfspace h with a corresponding generator $g(h)$ in the root of T_1 , we link $g(h)$ to a set of 2^d generators, instead of explicitly storing $P \cap h$. The 2^d generators are obtained by querying $q(h)$ in the data structure corresponding

to each child of the root. A second tree T_2 can be obtained by repeating the procedure above, from top to bottom, to every subtree of T_1 . The leaves of T_2 store the set of points contained in the corresponding cell, and no data points are stored anywhere else in the tree. Since the leaf cells are disjoint, and each generator in an internal node stores only $2^d = O(1)$ pointers, the storage space is $O(n + \log(1/\varepsilon)/\varepsilon^d)$.

In order to perform halfspace or halfbox queries, we need to follow the links all the way to the root. Consequently, the query time can be as high as $\Theta(1/\varepsilon^d)$, since we need to examine all leaf nodes inside the query halfspace. To bound the query time as a function of k , the number of points reported, we need to perform some kind of path compression in the tree.

First, we note that we do need to keep links to empty leaves or, more generally, links that can only lead to empty leaves. Let T_3 be the tree obtained by removing from T_2 all the links that are not in any path from the root data structure to a non-empty leaf. Second, we compress chains of single links. Let T_4 be the tree obtained from T_3 by directly linking a node v_1 to a leaf node v_2 , whenever v_2 is the only leaf node that can be reached from v_1 . We call T_4 a *reporting halfbox quadtree*.

The set of links obtained from a single query halfbox in T_4 defines a tree where each internal node has at least two children, and all leaf node store at least one distinct reported data point. The cost of traversing this tree when answering a query is proportional to the number of leaf nodes. The query time for halfspace and halfbox range searching to report k points is $O(k)$.

The tree T_2 can be preprocessed using the second halfspace range searching preprocessing algorithm from Section 5.2. The trees T_3 and finally T_4 can be

Range	Version	Query time
halfspace	reporting	$O(k)$
spherical	reporting	$O(k + 1/\varepsilon^{(d-1)/2})$
fat simplex	reporting	$O(k + 1/\varepsilon^{(d-2)} + \log(1/\varepsilon))$

Table 6.1: Query time of the reporting halfbox quadtree for different range shapes. The storage space and preprocessing time are $O(n + \log^{d+1}(1/\varepsilon)/\varepsilon^d)$.

obtained from T_2 in time proportional to the size of the original tree using tree traversal.

It is important to note, though, that because of error accumulation, as in the second preprocessing algorithm from Section 5.2, the query will be $O(\varepsilon \log(1/\varepsilon))$ -approximate. By adjusting ε accordingly, we conclude:

Theorem 6.8. *Let \mathcal{R} be a set of ranges for which the halfbox quadtree can answer ε -approximate range queries in $f(\varepsilon)$ time, in the semigroup version. A reporting halfbox quadtree can answer ε -approximate range reporting queries for \mathcal{R} in $O(f(\varepsilon) + k)$ query time, with $O(n + \log^{d+1}(1/\varepsilon)/\varepsilon^d)$ storage space and preprocessing time.*

The query time of the reporting halfbox quadtree for different range shapes is summarized in Table 6.1.

6.6 The Data Stream Model

Recently, a lot of attention has been given to problems where the data is too large to be stored statically. In these applications, the input is presented as an uncontrolled stream of data, which can only be examined once. Such problems

often arise from network monitoring, sensor networks, telecommunication, financial applications and others [14].

In the *data stream model* [14, 48], the input is too large to fit in memory, and possibly unbounded. Therefore the storage space should be independent of n (sometimes polylogarithmic functions of n are acceptable). Also, the data points are examined one at a time, in a single pass, while queries regarding the points that have already been seen need to be answered efficiently.

Exact range searching clearly requires $\Omega(n)$ storage for all reasonable sets of ranges, and approximate range searching in the relative model requires $\Omega(n)$ storage when the query ranges can be scaled to arbitrary small ranges, because the weight of each individual point can be recovered. Therefore, there is no hope to answer exact range searching or approximate range searching in the relative model in the data stream model. Suri, Tóth, and Zhou [53] consider approximate range counting in the data stream model, approximating the number of points inside the query region. In this section, we modify the halfbox quadtree to design a data structure that is even more efficient in the data stream model.

We use three parameters when evaluating a data structure in the data stream model: storage space, query time, and update time. The *update time* refers to the time to process each new point from the data stream. For a data structure to be *feasible* in the data stream model, all these three parameters must be independent of n .

The absolute model is naturally fit for data stream applications, since the storage space and the query time of the data structures are independent of n . However, the time to insert a new point in the halfbox quadtree can be

as large as the storage space. While this approach is still feasible, we would like to be able to insert points faster, in order to obtain a data stream data structure with small update time.

In the data structures we describe, the update process consists of simply inserting the new point in a set of generators. For all data structures we describe, we can trivially locate the generators containing a given point in $O(1)$ time per generator. Therefore, the update time can be calculated as the number of generators that contain the new point. Inserting a point in the semigroup halfspace range searching data structure from section 5.2 takes $\Omega(1/\varepsilon^d)$ time, because each data point is contained in $\Theta(1/\varepsilon^d)$ generators. Since the halfbox quadtree contains a halfspace range searching data structure, the time to insert a point in the halfbox quadtree cannot be any better. In this section, we modify the halfbox quadtree, in order to obtain a tradeoff between update time and query time.

First, we examine how many generators contain a given point p at each level of the halfbox quadtree. Since the quadtree boxes at each level of the quadtree are disjoint, only one quadtree box per level contains p . A quadtree box at level ℓ has size $2^{-\ell}$, and is associated with a halfspace data structure with $O(1/(2^\ell \varepsilon)^d)$ storage space. The total number of generators that contain p among all levels of the halfbox quadtree is

$$\sum_{\ell=0}^{\log(O(1/\varepsilon))} O(1/(2^\ell \varepsilon)^d) = O(1/\varepsilon^d).$$

Since the terms of the summation decrease geometrically, the top levels of the quadtree are the ones that contribute the most to the sum. If we prune

the top part of the halfbox quadtree, by removing all quadtree boxes of size larger than a parameter $s \in [\varepsilon, 1]$, the total number of generators that contain p becomes

$$\sum_{\ell=-\log s}^{\log(O(1/\varepsilon))} O(1/(2^\ell \varepsilon)^d) = O((s/\varepsilon)^d).$$

We call this data structure a s -pruned halfbox quadtree, or just *pruned halfbox quadtree*. The update time for the pruned halfbox quadtree is $O((s/\varepsilon)^d)$. The storage space for the pruned halfbox quadtree is $O(\log(1/\varepsilon)/\varepsilon^d)$, since the pruned halfbox quadtree is a subset of the halfbox quadtree.

To calculate the query time for the pruned halfbox quadtree, we need to examine the cost of replacing large halfboxes by smaller ones. We assume that the query is answered in a way that, for any range $R \in \mathcal{R}$, only a constant number of halfboxes in $g(R)$ correspond to the same quadtree box. The following theorem relates the query time for an s -pruned halfbox quadtree with the query time for a halfbox quadtree.

Theorem 6.9. *Let \mathcal{R} be a set of ranges for which the halfbox quadtree can answer ε -approximate range queries in $f(\varepsilon)$ time, in a way that, for any range $R \in \mathcal{R}$, only a constant number of halfboxes in $g(R)$ correspond to the same quadtree box. An s -pruned halfbox quadtree can answer ε -approximate semigroup range queries for \mathcal{R} in the data stream model in $O(f(\varepsilon) + 1/s^d)$ query time, with $O((s/\varepsilon)^d)$ update time and $O(\log(1/\varepsilon)/\varepsilon^d)$ storage space, for $s \in [\varepsilon, 1]$.*

Proof. The update time and storage space follow from the explanation in the text. We only need to prove the query time part of the theorem. Without loss of generality, we assume that $s = 1/2^k$ for some integer k . Let $g_{\leq s}(R)$ be the

subset of $g(R)$ containing only halfboxes of size at most s , and $g_{>s}(R)$ be the subset of $g(R)$ containing only halfboxes of size greater than s . The diameter of $\bigcup(g_{>s}(R))$ is $O(1)$, because $g_{>s}(R) \subseteq [0, 1]^d$. Since all halfboxes in $g_{>s}(R)$ have size greater than s , we can create $g'(R)$ such that $\bigcup(g'(R)) = \bigcup(g_{>s}(R))$, and $g'(R)$ contains only halfboxes of size exactly s . Using Lemma 2.3, and the fact that $g_{>s}(R)$ has only a constant number of halfboxes for each quadtree box, we conclude that $|g'(R)| = O(1/s^d)$. Therefore, we can keep the at most $f(\varepsilon)$ generators in $g_{\leq s}(R)$, and replace $g_{>s}(R)$ with $g'(R)$, totaling $O(f(\varepsilon) + 1/s^d)$ generators. \square

We can do slightly better than the Theorem 6.9 by using a non-pruned quadtree, as used in Section 4.2, together with the pruned halfbox quadtree. We call these two data structures together an *s-stream halfbox quadtree*, or just *stream halfbox quadtree*. The storage space for the stream halfbox quadtree is

$$O(1/\varepsilon^d + \log(1/\varepsilon)/\varepsilon^d) = O(\log(1/\varepsilon)/\varepsilon^d).$$

To calculate the update time, we only need to consider the additional work to update the generators in the standard quadtree. Each point p is in $O(\log(1/\varepsilon))$ generators, since a point is in only one generator for each level of the quadtree. Therefore, the update time for the stream halfbox quadtree is $O(\log(1/\varepsilon) + (s/\varepsilon)^d)$. The following theorem relates the query time for an *s-stream halfbox quadtree* with the query time for a halfbox quadtree.

Theorem 6.10. *Let \mathcal{R} be a set of convex ranges for which the halfbox quadtree can answer ε -approximate range queries in $f(\varepsilon)$ time, in a way that, for any*

range $R \in \mathcal{R}$, only a constant number of halfboxes in $g(R)$ correspond to the same quadtree box. An s -stream halfbox quadtree can answer ε -approximate range queries for \mathcal{R} in the data stream model in $O(f(\varepsilon) + 1/s^{d-1})$ query time, with $O((\log(1/\varepsilon) + s/\varepsilon)^d)$ update time and $O(\log(1/\varepsilon)/\varepsilon^d)$ storage space, for $s \in [\varepsilon, 1]$.

Proof. The update time and storage space follow from the explanation in the text. To calculate the query time for the pruned halfbox quadtree, we need to examine the cost of replacing large halfboxes by smaller ones, but consider the fact that quadtree boxes do not need to be replaced. Halfboxes that are not quadtree boxes are only useful for halfboxes that are intersected by the range boundary ∂R . Let $g(\partial R)$ denote the halfboxes in $g(R)$ that intersect ∂R .

Without loss of generality, we assume that $s = 1/2^k$ for some integer k . Let $g_{>s}(\partial R)$ be the subset of $g(\partial R)$ containing only halfboxes of size greater than s . Since all halfboxes in $g_{>s}(\partial R)$ have size greater than s , we can create $g'(\partial R)$ such that $\bigcup(g'(\partial R)) = \bigcup(g_{>s}(\partial R))$, and $g'(\partial R)$ contains only halfboxes of size exactly s . Using Lemma 2.4, and the fact that $g_{>s}(\partial R)$ has only a constant number of halfboxes for each quadtree box, we conclude that $|g'(\partial R)| = O(1/s^{d-1})$. Therefore, we can keep the at most $f(\varepsilon)$ generators in $g(R) \setminus g_{>s}(\partial R)$, and replace $g_{>s}(\partial R)$ with $g(\partial R)$, totaling $O(f(\varepsilon) + 1/s^{d-1})$ generators. \square

Note that all query algorithms presented in this text for the halfbox quadtree satisfy the conditions of Theorems 6.9, and 6.10. The query time of the stream halfbox quadtree for different range shapes is presented in Table 6.2.

Range	Version	Query time
halfspace	semigroup	$O(1/s^{d-1})$
spherical	semigroup	$O(1/\varepsilon^{(d-1)/2} + 1/s^{d-1})$
simplex	group	$O(1/\varepsilon^{(d-2)} + \log(1/\varepsilon) + 1/s^{d-1})$
fat simplex	semigroup	$O(1/\varepsilon^{(d-2)} + \log(1/\varepsilon) + 1/s^{d-1})$

Table 6.2: Query time of the stream halfbox quadtree for different range shapes, as a function of s . The storage space is $O(\log(1/\varepsilon)/\varepsilon^d)$, and the update time is $O(\log(1/\varepsilon) + (s/\varepsilon)^d)$.

We may want to have the update time equal or similar to the query time. For the case of halfspace range searching, if we set $s = \varepsilon^{\frac{1}{2d-1}}$, then both the query and update times are

$$O\left(\frac{1}{\varepsilon^{\frac{d-1}{2d-1}}}\right) = O\left(\frac{1}{\varepsilon^{\frac{1}{2} - \frac{1}{4d-2}}}\right) = o\left(\frac{1}{\sqrt{\varepsilon}}\right)$$

independently of the value of d .

For the case of spherical range searching, if we set $s = \varepsilon^{(d+1)/2d}$, then both the query and update times are $O(1/\varepsilon^{\frac{d}{2} - \frac{1}{2d}}) = o(1/\varepsilon^{d/2})$.

6.7 The Relative Error Model

In the *relative error model* (or simply *relative model*), it is assumed that the range shape R is bounded, and points lying within distance $\varepsilon \cdot \text{diam}(R)$ of the boundary of the range may or may not be included. An example different query ranges and their fuzzy boundaries in the relative model is illustrated in Figure 6.5. Range searching in the relative model has been studied in [8, 9, 10, 12]. In this section, we use the halfspace range searching data structure

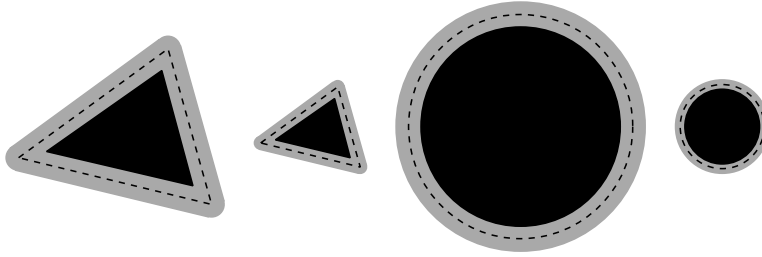


Figure 6.5: Fuzzy boundaries of different query ranges in the relative model.

from Theorem 5.2 to build the relative halfbox quadtree, a data structure that is analogous to halfbox quadtree, but works in the relative error model.

The best upper bounds previously known for approximate range searching in the relative error model with general semigroups and constant dimension are the following. Arya and Mount [12] present a data structure which answers arbitrary convex range queries (using the unit-cost test assumption) in $O(\log(n) + 1/\varepsilon^{d-1})$ query time, with $O(n)$ storage space, and $O(n \log(n))$ preprocessing time. Arya, Malamatos, and Mount [8] present a space-time tradeoff for spherical ranges with $O(\log(n\gamma) + 1/(\varepsilon\gamma)^{d-1})$ query time, $O(n\gamma^d \log(1/\varepsilon))$ storage space, and $O(n\gamma^d \log(n/\varepsilon) \log(1/\varepsilon))$ preprocessing time, for $1 \leq \gamma \leq 1/\varepsilon$. Arbitrary smooth convex ranges are also studied in [9], but an upper bound is only given for the idempotent case.

In this section, we show how the relative halfbox quadtree gives a space-time tradeoff for smooth convex ranges which improves the best previous results for spherical range searching by logarithmic terms in the storage space, query time, and preprocessing time. We also show how the relative halfbox quadtree is the first approximate simplex range searching data structure (in the group version) to improve over the $O(1/\varepsilon^{d-1})$ query time for arbitrary convex ranges. If the query simplex is fat, the result holds in the semigroup

Range	Remark	Storage	Query time
Smooth	tradeoff, $1 \leq \gamma \leq 1/\sqrt{\varepsilon}$	$O(n\gamma^d)$	$O(\log n + 1/(\varepsilon\gamma)^{d-1})$
	high storage	$O(n/\varepsilon^{d/2})$	$O(\log n + 1/\varepsilon^{(d-1)/2})$
	low storage	$O(n)$	$O(\log n + 1/\varepsilon^{d-1})$
Simplex	tradeoff, $1 \leq \gamma \leq 1/\varepsilon^{1/(d-1)}$	$O(n\gamma^d)$	$O(\log n + \log(1/\varepsilon) + 1/(\varepsilon\gamma)^{d-1})$
	high storage	$O(n/\varepsilon^{\frac{d}{d-1}})$	$O(\log n + \log(1/\varepsilon) + 1/\varepsilon^{d-2})$
	low storage	$O(n)$	$O(\log n + 1/\varepsilon^{d-1})$

Table 6.3: Complexities of the relative halfbox quadtree for different range shapes. Simplex results work in the group version for general simplices and in the semigroup version for fat simplices. The preprocessing time is equal to the storage space with an additive term of $O(n \log n)$.

version. We summarize our results in Table 6.3.

Let Δ be the diameter of the query range R . A relative model query is denoted by $q_{\varepsilon\Delta}(R)$ because it is equivalent to an $(\varepsilon\Delta)$ -approximate query in the absolute model.

Let $\gamma \geq 1$ be a parameter. We define a *relative halfbox quadtree* as a compressed quadtree T , indexed by a finger tree T' , where each quadtree box Q of size δ stored in T is associated with an absolute model (δ/γ) -approximate halfspace range searching data structure from Theorem 5.2. The general idea is that halfboxes can be used to approximate a range R in the same manner as in the absolute model, as long as $\delta/\gamma \leq \Delta\varepsilon$.

A (δ/γ) -approximate halfspace range searching data structure for a quadtree box of size δ takes $O(\gamma^d)$ storage space. Since T has $O(n)$ nodes, the total storage space for the relative halfbox quadtree is $O(n\gamma^d)$.

To preprocess T , we start by building a compressed quadtree and a finger tree in $O(n \log n)$ time (see Section 2.2 for details). The leaves of the compressed quadtree contain a single data point. Therefore, we can build a halfspace range searching data structure associated with the leaf nodes in

$O(\gamma^d)$ time for each leaf node. The data structure for an internal node v can be built in $O(\gamma^d)$ time by performing at most 2^d queries to the children of v for each entry in the lookup table, in the same way as the preprocessing algorithm from Section 5.2. Note that the total error accumulation in the root of the tree is bounded by a factor of 2, because the quadtree boxes of the children of a node v have at most half the size of v_\square . Consequently, the preprocessing time for a relative halfbox quadtree is $O(n \log n + n\gamma^d)$, which is optimal.

Given a parameter α , a convex shape R is α -smooth if, for any point x in the boundary of R , there is a ball $B \subseteq R$ of diameter $\alpha \text{diam}(R)$ that touches x . A range is *smooth* if it is α -smooth for some constant α . We say that a hyperplane h is tangent to R at point x if it is tangent to a ball $B \subseteq R$ of radius $\alpha \text{diam}(R)$ that touches x . The following lemma is the key for answering smooth range queries efficiently.

Lemma 6.11. *Let R be an α -smooth range of diameter Δ , ε an approximation parameter, and Q a quadtree box of diameter δ . If $\delta \leq \Delta\sqrt{\alpha\varepsilon}$, then any halfspace h that is tangent to R at a point $x \in Q$ $\Delta\varepsilon$ -approximates R within Q .*

Proof. Let x be a point in the boundary of R and inside Q . Let $B \subseteq R$ be a ball of radius $\alpha\Delta$ that touches x and h be the hyperplane tangent to B at point x (Figure 6.6). Note that the boundary of R must be between h and B within Q , because R is convex and smooth. It follows from Lemma 2.2 that h ε -approximates R within Q . \square

In order to answer queries efficiently, we need to make an assumption that is stronger than the *unit-cost-test assumption* used in [9, 12]. We assume that,

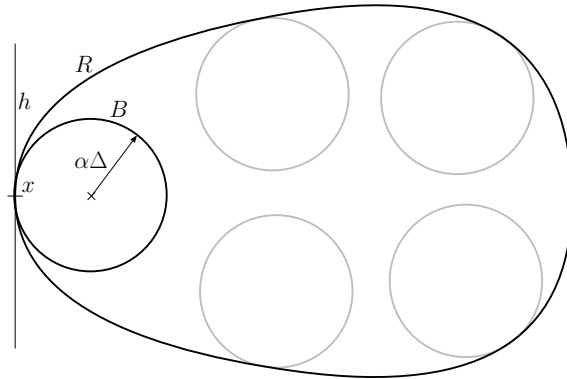


Figure 6.6: Smooth range in the proof of Lemma 6.11.

given a smooth convex range R and a quadtree box Q , in constant time we can determine whether $Q \subseteq R$, is $Q \cap R = \emptyset$, or neither, and, in the latter case, we can also find in constant time a hyperplane that is tangent to R at some point $x \in Q$. Note that this assumption is easily satisfied for spherical ranges.

Let $\delta > 0$ be a constant, and R be a query range of diameter Δ . Let a be a valid diameter for a quadtree box, with a between $\delta\Delta$ and $\delta\Delta/2$. To answer a range query $q_{\Delta\epsilon}(R)$, we first determine the set A of quadtree boxes of diameter a that intersect R . By Lemma 2.3, we have $|A| = O(1)$. Let V be the set formed by the result of cell queries for each element of A . Note that $|V| = |A| = O(1)$, and V can be computed in $O(\log(n))$ time. The query is answered by performing the following procedure for each $v \in V$, and summing the results.

1. If $v_{\square} \cap R = \emptyset$, then return 0.
2. If $v_{\square} \subset R$, then return $w(v)$.

3. If $|\dot{v}| = 1$, then we verify whether the point stored in v is contained in R and return the weight of the point or 0, accordingly.
4. If $diam(v_{\square})/\gamma \leq \Delta\varepsilon/2$ and $diam(v_{\square}) \leq \Delta\sqrt{\alpha\varepsilon/2}$, then we determine a halfspace h that is tangent to R inside v_{\square} , and return the precomputed $q_{diam(v_{\square})/\gamma}(h \cap v_{\square})$.
5. Otherwise, we return the sum of the recursive calls of the procedure for each child of v .

We only need to show the correctness of Step 4, since the other steps are clearly correct. To show that Step 4 is correct, we note that, by Lemma 6.11, the halfspace h $(\Delta\varepsilon/2)$ -approximates R within v_{\square} . Also, v is associated with a $(\Delta\varepsilon/2)$ -approximate halfspace data structure, because $diam(v_{\square})/\gamma \leq \Delta\varepsilon/2$. Adding both approximation errors, we obtain a $(\Delta\varepsilon)$ -approximation.

To analyze the query time when α is a constant, we note that a recursive call is only performed when v_{\square} intersects the boundary of R and

$$diam(v_{\square}) = O(\Delta\varepsilon\gamma + \Delta\sqrt{\varepsilon}).$$

Using Lemma 2.4, and summing the number of recursive calls, we conclude that the query takes $O(1/(\varepsilon\gamma)^{d-1} + 1/\varepsilon^{(d-1)/2})$ time. Note that there is no advantage in setting $\gamma > 1/\sqrt{\varepsilon}$. The following theorem summarizes the result.

Theorem 6.12. *The relative halfbox quadtree with $1 \leq \gamma \leq 1/\sqrt{\varepsilon}$ is an ε -approximate range searching data structure for smooth ranges, in the relative model, with $O(n\gamma^d)$ storage space, $O(\log n + 1/(\varepsilon\gamma)^{d-1})$ query time, and $O(n \log n + n\gamma^d)$ preprocessing time.*

In the high storage version, when $\gamma = 1/\sqrt{\varepsilon}$, Theorem 6.12 gives an approximate smooth range searching data structure with $O(n/\varepsilon^{d/2})$ storage space, $O(\log n + 1/\varepsilon^{(d-1)/2})$ query time, and $O(n \log n + n/\varepsilon^{d/2})$ preprocessing time.

The relative halfbox quadtree can also be used to answer simplex range queries, as long as either subtraction is allowed or the query simplex is fat. We first determine a set of $O(1)$ vertices V , in $O(\log n)$ time, in the same way as in spherical range searching. Then, the simplex range query is answered by performing the following procedure for each $v \in V$, and summing the results.

1. If $v_{\square} \cap R = \emptyset$, then we return 0.
2. If $v_{\square} \subset R$, then we return the precomputed $w(Q) = \sum_{p \in P \cap Q} w(p)$.
3. If $|\dot{v}| = 1$, then we verify whether the point stored in v is contained in R and return the weight of the point or 0, accordingly.
4. If $\text{diam}(v_{\square}) \leq \Delta\varepsilon$, then we verify whether R contains the center of v_{\square} and answer accordingly.
5. *Group version:* If $\text{diam}(v_{\square})/\gamma < \Delta\varepsilon$ and v_{\square} does not contain any $(d-2)$ -faces of R , then there is a set H of at most $d+1$ pairwise disjoint halfspaces that form the complement of R . We return $w(v) - \sum_{h \in H} q_{\Delta\varepsilon}(h \cap v_{\square})$, where $h \cap v_{\square}$ is a halfbox, and $q_{\varepsilon}(\cdot)$ is the result of the approximate query using the data structure associated with v .

Fat simplex version: If $\text{diam}(v_{\square})/\gamma < \Delta\varepsilon$ and v_{\square} contains only one $(d-1)$ -face h of R , then we return $q_{\Delta\varepsilon}(h \cap v_{\square})$, where $h \cap v_{\square}$ is a halfbox, and $q_{\varepsilon}(\cdot)$ is the result of the approximate query using the data structure associated with v .

6. Otherwise, we return the sum of $q(c, R)$ for all children c of v in the compressed quadtree.

To analyze the query time for the group version, we note that a recursive call is only performed when either (i) v_\square intersects the boundary of R and $\text{diam}(v_\square) = O(\Delta\varepsilon\gamma)$, or (ii) v_\square intersects a $(d-2)$ -face of R and $\text{diam}(v_\square) \leq \Delta\varepsilon$. Using Lemmas 2.4 and 2.5, and summing the number of recursive calls, we conclude that the query takes $O(\log(1/\varepsilon) + 1/(\varepsilon\gamma)^{d-1} + 1/\varepsilon^{d-2})$ time. Note that there is no advantage in setting $\gamma > 1/\varepsilon^{1/(d-1)}$.

To analyze the query time for the fat simplex semigroup version, we note that a recursive call is only performed when either (i) v_\square intersects the boundary of R and $\text{diam}(v_\square) = O(\Delta\varepsilon\gamma)$, or (ii) v_\square intersects more than one $(d-1)$ -faces of R and $\text{diam}(v_\square) \leq \Delta\varepsilon$. Using Lemmas 2.4 and 2.6, and summing the number of recursive calls, we conclude that the query takes $O(\log(1/\varepsilon) + 1/(\varepsilon\gamma)^{d-1} + 1/\varepsilon^{d-2})$ time. Note that there is no advantage in setting $\gamma > 1/\varepsilon^{1/(d-1)}$. The following theorem summarizes the result.

Theorem 6.13. *The relative halfbox quadtree with $1 \leq \gamma \leq 1/\varepsilon^{1/(d-1)}$ is an ε -approximate range searching data structure for simplex ranges, in the relative model and group version, with $O(n\gamma^d)$ storage space, $O(\log n + \log(1/\varepsilon) + 1/(\varepsilon\gamma)^{d-1})$ query time, and $O(n \log n + n\gamma^d)$ preprocessing time. The result holds for general semigroups if the query simplex is fat.*

In the high storage version, when $\gamma = 1/\varepsilon^{1/(d-1)}$, Theorem 6.13 gives a data structure with $O(n/\varepsilon^{d/(d-1)})$ storage space, $O(\log n + \log(1/\varepsilon) + 1/\varepsilon^{d-2})$ query time, and $O(n \log n + n/\varepsilon^{d/(d-1)})$ preprocessing time.

Chapter 7

Conclusion

In this dissertation, we have introduced approximate range searching data structures for several fundamental range spaces. Our work differs from previous works in that we use the absolute model (in contrast to the relative model used in [8, 9, 10, 12]), and we consider the dimension d to be a constant (in contrast to spaces of high dimensions considered in [26]). Most data structures presented in this dissertation are not only more efficient, but are also much simpler than both their exact and relative model counterparts. We also applied the absolute model data structures for different problems and models.

7.1 Absolute Model Data Structures

The first data structure we presented answers orthogonal range queries. Orthogonal ranges are the only widely studied range shapes for which exact queries can be answered in polylogarithmic time with near linear space [20, 21]. Nevertheless, more efficient and simpler solutions for the approximate ver-

sion exist, especially in the group version. The approximate orthogonal range searching data structure is the only data structure in this dissertation based on the point approximation technique for the semigroup and group versions. While the point approximation technique can be used with most range shapes, the query time and storage space obtained this way are generally the same as to the ones for the exact version of the problem, with n replaced by $1/\varepsilon^d$. For the case of approximate orthogonal range searching, the query time can be improved because partial-sum data structures can be used [28, 57].

The simplest data structure we presented is the quadtree used to answer queries with arbitrary convex ranges. Because of the unbounded complexity of arbitrary convex ranges, we used the unit-cost test assumption as in [12]. The data structure is an important building block for the halfbox quadtree. It is also the only approximate range searching data structure in this dissertation whose storage space can be bounded by $O(n)$ regardless of the value of ε .

The data structures for approximate halfspace range searching are among the most efficient data structures presented in this dissertation. This is surprising considering the high complexity of the exact version of the problem [10, 18]. In the semigroup version, a simple table lookup technique can be used to answer halfspace range searching queries in $O(1)$ time with $O(1/\varepsilon^d)$ storage space. This data structure is an important building block for the halfbox quadtree. Other approximate halfspace range searching data structures are presented for idempotent and emptiness queries. If the semigroup is idempotent, a space-time tradeoff can be obtained, reducing the space to as low as $O(1/\varepsilon^{(d+1)/2})$, while keeping the product of the storage space and the query time equal to $O(1/\varepsilon^d)$. For the emptiness version, the storage space goes as

low as $O(1/\varepsilon^{(d-1)/2})$, while the product of the storage space and the query time is equal to $O(1/\varepsilon^{d-1})$. Alternatively, the point approximation technique can be used, together with ε -kernels [3, 19] and exact halfspace emptiness data structures [30, 41, 46, 51], to obtain data structures for the approximate emptiness version, which are especially efficient for $d \leq 3$.

Approximate spherical range searching is widely studied in the relative model [8, 9, 10]. We combine our approximate halfspace range searching data structure with our data structure for convex ranges to obtain the halfbox quadtree. The halfbox quadtree efficiently answer approximate spherical queries in the absolute model. Additional generators can be included in the data structure, in order to reduce the query time.

Simplex range searching is widely studied in the exact version [2, 40, 42, 43]. The halfbox quadtree provides an efficient data structure to answer approximate simplex queries. The use of subtraction is necessary for our query algorithm unless the query simplex is fat.

7.2 Applications

The first related problem we presented is the range sketching problem. This problem tries to fill the gap between range counting and range reporting. The data structure is obtained by threading a finger tree with a compressed quadtree. The same technique is used in the relative halfbox quadtree.

We showed how idempotent absolute error model halfspace range searching can be applied to provide a tight upper bound for exact halfspace range searching, by properly setting the value of ε as a function of n . This exact

data structure is defined in the semigroup arithmetic model, and we assume that the data points are uniformly distributed inside the unit hypercube. The data structure matches the lower bound proved in [18] up to logarithmic factors. The theoretical importance of the data structure relies on the fact that uniform distribution and the semigroup arithmetic model are also assumed in the lower bound proved in [18].

We showed how to use the halfbox quadtree to answer approximate nearest neighbor queries, approximate range reporting queries, approximate range queries in the data stream model [14, 48, 53], and approximate range queries in the relative model. Different tools were necessary for these variations, but the underlying techniques remain the same. The most interesting of these results is the relative halfbox quadtree, since there are several previous data structures for approximate range searching in the relative model [8, 9, 10, 12].

The relative halfbox quadtree combines a compressed quadtree, a finger tree, and a data structure for halfspace range searching in the absolute model. The strong points of the relative halfbox quadtree when compared to other existing structures are: (i) simplicity, (ii) faster query time (by a logarithmic term), (iii) reduced storage space (by a logarithmic factor), (iv) optimal preprocessing time, and (v) the ability to handle not only spherical ranges, but also smooth ranges, simplex ranges, and polyhedral ranges.

7.3 Future Work

We focused this work towards structures with relatively low storage space (mostly $\tilde{O}(1/\varepsilon^d)$). Note that in the absolute model any range query can be

answered in $O(1)$ time with $O(1/\varepsilon^d)$ storage space, ignoring the time needed to determine the set of generators. A possible direction for future work consists of determining the minimum storage space necessary to answer queries in $O(1)$ time, for different range shapes, as well as finding new space-time tradeoffs.

We showed how to construct an exact halfspace range searching data structure, that uses idempotence to improve over the most efficient exact data structure previously known. The data structure is defined in the semigroup arithmetic model [10, 18, 22] and matches the lower bound proved in [18] up to logarithmic factors. The theoretical importance of the data structure relies on the fact that uniform distribution and the semigroup arithmetic model are also assumed in the lower bound proved in [18]. Therefore, we open some important theoretical and practical questions: Is the average case complexity for uniformly distributed data strictly lower than the worst case complexity? Does the semigroup arithmetic model allow more efficient idempotent halfspace range searching data structures than the real RAM model? An improved data structure that worked in the real RAM model would be of practical interest, even if it relied on the uniform distribution of the points.

We showed how to use $O(\log(1/\varepsilon))$ approximate spherical emptiness queries to answer approximate nearest neighbor queries in the absolute error model. The approximate nearest neighbor query time can certainly be reduced by increasing the storage space, but precise space-time tradeoffs still need to be determined.

There are several non-trivial lower bounds for exact range searching [10, 18, 22, 23]. There are also some non-trivial lower bound results for approximate range searching in the relative model [9, 10]. It would be interesting

to investigate lower bounds for approximate range searching in the absolute model.

The following weaknesses of the relative halfbox quadtree are important topics for future research on approximate range searching in the relative model. First, the space-time tradeoff for spherical range searching is limited to query times that are not faster than $\Theta(\log n + 1/\varepsilon^{(d-1)/2})$. While this query time is optimal for arbitrary smooth ranges [9], it can be improved for spherical ranges, at the cost of additional storage space [8].

Second, we do not know how to improve the space-time tradeoff of the relative halfbox quadtree for the case of idempotent semigroups. Significantly faster data structures for approximate idempotent spherical range searching exist [10]. The data structure for approximate idempotent halfspace range searching in the absolute model could possibly be used to obtain an idempotent version of the relative halfbox quadtree.

Finally, the fastest data structures for approximate range searching over arbitrary groups and semigroups in the relative model are still significantly slower than the existing lower bounds. The best lower bound for approximate spherical range searching in the relative model over integral semigroups is $\Omega((n/m)^{1-1/d}/\varepsilon^{d-5})$ query time with m storage space [10]. The best lower bound for approximate simplex range searching in the relative model over integral semigroups is $\Omega(1/\varepsilon^{d-2-2fd})$ query time with $n/\varepsilon^{f^2d^2}$ storage space and $0 \leq f \leq 1$ [9].

Bibliography

- [1] Peyman Afshani and Timothy M. Chan. On approximate range counting and depth. In *SCG '07: Proceedings of the twenty-third annual symposium on Computational geometry*, pages 337–343, 2007.
- [2] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. In Bernard Chazelle, Jacob Goodman, and Richard Pollack, editors, *Advances in Discrete and Computational Geometry*, pages 1–56. American Mathematical Society, Providence, 1998.
- [3] Pankaj K. Agarwal, Sariel Har-Peled, and Kasturi R. Varadarajan. Geometric approximation via coresets. In Jacob E. Goodman, János Pach, and Emo Welzl, editors, *Combinatorial and Computational Geometry*, MSRI Publications. Cambridge Univ. Press, 2005.
- [4] Pankaj K. Agarwal and Jiří Matoušek. On range searching with semialgebraic sets. In *MFCS '92: Proceedings of the 17th International Symposium on Mathematical Foundations of Computer Science*, pages 1–13, 1992.
- [5] Boris Aronov, Sariel Har-Peled, and Micha Sharir. On approximate half-space range counting and relative epsilon-approximations. In *SCG '07: Proceedings of the twenty-third annual symposium on Computational geometry*, pages 327–336, 2007.
- [6] Sunil Arya and Theodoros Malamatos. Linear-size approximate Voronoi diagrams. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 147–155, 2002.
- [7] Sunil Arya, Theodoros Malamatos, and David M. Mount. Space-efficient approximate Voronoi diagrams. In *STOC '02: Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, pages 721–730, 2002.

- [8] Sunil Arya, Theocharis Malamatos, and David M. Mount. Space-time tradeoffs for approximate spherical range counting. In *16th Ann. ACM-SIAM Symposium on Discrete Algorithms, (SODA '05)*, pages 535–544, 2005.
- [9] Sunil Arya, Theocharis Malamatos, and David M. Mount. The effect of corners on the complexity of approximate range searching. In *Proceedings of the 22nd ACM Symp. on Computational Geometry (SoCG'06)*, pages 11–20, 2006.
- [10] Sunil Arya, Theocharis Malamatos, and David M. Mount. On the importance of idempotence. In *Proc. 38th ACM Symp. on Theory of Computing (STOC'06)*, pages 564–573, 2006.
- [11] Sunil Arya and David M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 271–280, 1993.
- [12] Sunil Arya and David M. Mount. Approximate range searching. *Comput. Geom. Theory Appl.*, 17(3-4):135–152, 2000.
- [13] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.
- [14] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of ACM Principles of Database Systems*, pages 1–16, 2002.
- [15] Julien Basch, Leonidas J. Guibas, and Gurumurthy D. Ramkumar. Reporting red-blue intersections between two sets of connected line segments. In *Proceedings of the 4th Annual European Symposium on Algorithms*, volume 1136 of *Lecture Notes Comput. Sci.*, pages 302–319. Springer-Verlag, 1996.
- [16] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Commun, ACM*, 18:509–517, 1975.
- [17] Jon L. Bentley. Decomposable searching problems. *Inform. Process. Lett.*, 8:244–251, 1979.
- [18] Hervé Brönnimann, Bernard Chazelle, and János Pach. How hard is half-space range searching. *Discrete & Computational Geometry*, 10:143–155, 1993.

- [19] Timothy M. Chan. Faster core-set constructions and data stream algorithms in fixed dimensions. In *Proceedings of the 20th ACM Symp. on Computational Geometry (SoCG'04)*, pages 152–159, 2004.
- [20] Bernard Chazelle. Filtering search: a new approach to query answering. *SIAM J. Comput.*, 15(3):703–724, 1986.
- [21] Bernard Chazelle. Functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.
- [22] Bernard Chazelle. Lower bounds on the complexity of polytope range searching. *J. Amer. Math. Soc.*, 2:637–666, 1989.
- [23] Bernard Chazelle. Lower bounds for orthogonal range searching: part ii. the arithmetic model. *J. ACM*, 37(3):439–463, 1990.
- [24] Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9(2):145–158, 1993.
- [25] Bernard Chazelle. *The discrepancy method: randomness and complexity*. Cambridge University Press, New York, NY, USA, 2000.
- [26] Bernard Chazelle, Ding Liu, and Avner Magen. Approximate range searching in higher dimension. In *Proceedings of the 16th Canadian Conference on Computational Geometry (CCCG'04)*, pages 154–157, 2004.
- [27] Bernard Chazelle and Jiří Matoušek. On linear-time deterministic algorithms for optimization problems in fixed dimension. *J. Algorithms*, 21(3):579–597, 1996.
- [28] Bernard Chazelle and Burton Rosenberg. Computing partial sums in multidimensional arrays. In *SCG '89: Proceedings of the fifth annual symposium on Computational geometry*, pages 131–139, 1989.
- [29] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwartzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2000.
- [30] David P. Dobkin, John Hershberger, David G. Kirkpatrick, and Subhash Suri. Implicitly searching convolutions and computing depth of collision. In *Proceedings of the International Symposium on Algorithms (SIGAL'90)*, pages 165–180, 1990.
- [31] Herbert Edelsbrunner. *Algorithms in combinatorial geometry*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

- [32] David Eppstein, Michael T. Goodrich, and Jonathan Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*, pages 296–305, New York, NY, USA, 2005. ACM Press.
- [33] Guilherme D. da Fonseca. Approximate range searching: The absolute model. In *Proceedings of the 10th International Workshop on Algorithms and Data Structures (WADS'07)*, pages 2–14, 2007. Journal version submitted to *Comput. Geom. Theory Appl.*
- [34] Guilherme D. da Fonseca, Sunil Arya, and David M. Mount. Approximate range searching and range sketching. Submitted to the 24th ACM Symp. on Computational Geometry (SoCG'08).
- [35] Sariel Har-Peled. Geometric approximation algorithms. <http://valis.cs.uiuc.edu/~sariel/teach/notes/aprx/>.
- [36] Sariel Har-Peled. A replacement for Voronoi diagrams of near linear size. In *FOCS*, pages 94–103, 2001.
- [37] D.T. Lee and C.K. Wong. Quintary trees: a file structure for multidimensional database systems. *ACM Trans. Database Syst.*, 5:339–353, 1980.
- [38] Chi-Yuan Lo and W.L. Steiger. An optimal time algorithm for ham-sandwich cuts in the plane. In *Proc. Second Canad. Conf. on Computational Geom.*, pages 5–9, 1990.
- [39] George S. Lueker. A data structure for orthogonal range queries. *IEEE Sympos. Found. Comput. Sci.*, pages 28–34, 1978.
- [40] Jiří Matoušek. Efficient partition trees. *Discrete & Computational Geometry*, 8:315–334, 1992.
- [41] Jiří Matoušek. Reporting points in halfspaces. *Comput. Geom. Theory Appl.*, 2(3):169–186, 1992.
- [42] Jiří Matoušek. Range searching with efficient hierarchical cutting. *Discrete & Computational Geometry*, 10:157–182, 1993.
- [43] Jiří Matoušek. Geometric range searching. *ACM Computing Surveys*, 26(4):421–461, 1994.
- [44] Jiří Matoušek. Approximations and optimal geometric divide-and-conquer. *Journal of Computer and System Sciences*, 50:203–208, 1995.

- [45] Jiří Matoušek. Derandomization in computational geometry. *J. Algorithms*, 20(3):545–580, 1996.
- [46] Jiří Matoušek and Otfried Schwarzkopf. On ray shooting in convex polytopes. *Discrete & Computational Geometry*, 10:215–232, 1993.
- [47] Nimrod Megiddo. Partitioning with two lines in the plane. *J. Algorithms*, 6(3):430–433, 1985.
- [48] S. Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers, 2005.
- [49] Mark H. Overmars. Efficient data structures for range searching on a grid. *J. Algorithms*, 9(2):254–275, 1988.
- [50] Mark H. Overmars and A. Frank van der Stappen. Range searching and point location among fat objects. *J. Algorithms*, 21(3):629–656, 1996.
- [51] Franco P. Preparata and Michael I. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.
- [52] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., 2005.
- [53] Subhash Suri, Csaba D. Tóth, and Yunhong Zhou. Range counting over multidimensional data streams. *Discrete & Computational Geometry*, 36(4):633–655, 2006.
- [54] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.
- [55] Vladimir N. Vapnik and Alexey Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Prob. and its Appl.*, 16(2):265–280, 1971.
- [56] Dan E. Willard. The super-B-tree algorithm. Technical Report TR-03-79, Aiken Comput. Lab., Harvard Univ., 1979.
- [57] Andrew C. Yao. Space-time tradeoff for answering range queries (extended abstract). In *Proc. 14th ACM Symp. on Theory of Computing (STOC'82)*, pages 128–136, 1982.

Index

- α -fat, 26
- α -smooth, 104
- ε -approximation, 41
- ε -cutting, 42
- ε -generates, 16
- ε -hull, 75
- ε -kernel, 75
- ε -net, 41
- x_d -intercept, 63

- absolute counting error, 4
- absolute error model, 5
- absolute model, 5
- approximate nearest neighbor, 86, 87
- approximate Voronoi diagram, 46
- ASA model, 16

- bounded VC-dimension, 41

- canonical triangulation, 42
- cell, 18
- cell query, 22, 60
- column, 73
- compressed quadtree, 20
- convex range, 3
- convex range searching, 56
- crossing number, 37
- cutting, 42

- data points, 1
- data stream model, 7, 12, 96
- dual, 39
- dual space, 39
- duality, 39

- external approximation, 17

- fat, 27, 91
 - simplex, 91
- fat simplex, 3
- finger tree, 21
- fuzzy boundary, 5

- generator, 16
- geometric duality, 39
- group version, 2, 16

- half quadtree box, 82
- halfbox, 82
 - size, 82
- halfbox quadtree, 81, 82
 - emptiness, 87
 - preprocessing, 82
 - pruned, 98
 - relative, 103
 - reporting, 94
 - stream, 99
- halfspace range, 3, 62
- halfspace range searching
 - emptiness, 74
 - exact, 35, 38, 78
 - idempotent, 70
 - semigroup, 66
- ham sandwich cut, 35
- hierarchical cutting, 43

- idempotent semigroup, 2, 16, 46, 65
- idempotent version, 2, 16
- integral semigroup, 47, 65

- internal approximation, 17
- kd-tree, 30
- multi-level data structure, 33, 53
- nearest neighbor, 46, 86
 - approximate, 46
- normal vector, 25
- one-reporting query, 88
- orthogonal range, 3, 30, 49
- orthogonal range reporting, 55
- orthogonal range searching, 49
 - exact, 30
 - group, 51
 - semigroup, 52
- packing lemma, 23
- partial sum, 50
- partition tree, 18
- point approximation, 49
- preprocessing, 17
 - halfbox quadtree, 82
 - halfspace, 68, 69, 85
 - relative halfbox quadtree, 103
- primal space, 39
- pruned halfbox quadtree, 98
- quadtree, 19, 82
- quadtree box, 20, 82
- query, 1
- query time, 17
- range, 1
 - counting, 1
 - emptiness, 2, 74
 - query, 1
 - reporting, 1, 93
 - searching, 1
 - sketching, 59
 - space, 1, 40
- range tree, 32
- relative counting error, 4
- relative error model, 5, 44, 101
- relative halfbox quadtree, 103
- relative model, 5, 44, 101
- reporting halfbox quadtree, 94
- semigroup version, 1, 16
- separator, 21
- set system, 40
- simplex range, 3, 35, 38, 89
- simplex range searching, 89, 108
 - exact, 35, 38
- simplicial partition, 37
- slope, 63
- smooth, 3, 104
- smooth range searching, 106
- spherical range, 3, 83
- spherical range searching, 83, 106
- storage space, 17
- stream halfbox quadtree, 99
- unit-cost test assumption, 44, 56
- unit-cost-test assumption, 104
- update time, 96
- VC-dimension, 41