ABSTRACT

Title of dissertation:	DYNAMIC DATA STRUCTURES FOR GEOMETRIC SEARCH AND RETRIEVAL
	Eunhui Park, 2013
Dissertation directed by:	Professor David M. Mount Department of Computer Science

Data structures for geometric search and retrieval are of significant interest in areas such as computational geometry, computer graphics, and computer vision. The focus of this dissertation is on the design of efficient dynamic data structures for use in approximate retrieval problems in multidimensional Euclidean space. A number of data structures have been proposed for storing multidimensional point sets. We will focus on quadtree-based structures. Quadtree-based structures possess a number of desirable properties, and they have been shown to be useful in solving a wide variety of query problems, especially when approximation is involved.

First, we introduce two dynamic quadtree-based data structures for storing a set of points in space, called the *quadtreap* and the *splay quadtree*. The quadtreap is a randomized variant of a quadtree that supports insertion and deletion and has logarithmic height with high probability. The splay quadtree is also a quadtree variant, but this data structure is self-adjusting, that is, it rebalances itself depending on the access pattern. It supports efficient insertion and deletion in the amortized sense.

We also study how to dynamically maintain an important geometric structure, called the *well-separated pair decomposition* (WSPD). A set of n points defines $O(n^2)$ pairs. Callahan and Kosaraju introduced the WSPD as a concise method for approximating the set of all pairs by using only $O(n)$ subsets of pairs that are spatially well-separated from each other. We present the first *output sensitive algorithm* for maintaining a WSPD for a dynamic point set, that is, one in which the running time depends on the actual number of newly created (or newly destroyed) pairs.

Finally, we consider maintaining a data structure for points in motion. Motion is often presented incrementally in discrete time steps, and future motion may not be predictable from the past. This is called the *black-box model*. We present efficient online algorithms for maintaining two important geometric data structures for moving points in the black-box model, namely *nets* and *net trees*. We establish the efficiency of these online algorithms through a competitive analysis.

DYNAMIC DATA STRUCUTRES

FOR

GEOMETRIC SEARCH AND RETRIEVAL

by

Eunhui Park

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
2013

Advisory Committee:
Professor David M. Mount, Chair/Advisor
Professor Ramani Duraiswami
Professor Samir Khuller
Professor Sung Lee
Professor Hanan Samet

© Copyright by
Eunhui Park
2013

To my husband, Chanhyun and my son, Taewan.

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost I'd like to express my sincere gratitude to my advisor, Professor David Mount for his efforts to lead me to a good researcher. He encourages my academic motivation and guides me to right directions whenever I struggled with difficult problems. All I have learned from him will benefit me for my whole life. I also would like to give my special thanks to Professor Samir Khuller. He gave me sincere and considerate advice for my study and my career.

I would like to thank Professor Hanan Samet, Professor Ramani Duraiswami, and Professor Sung Lee for serving on my thesis committee and for spending their invaluable time reviewing my dissertation. In particular, I deeply appreciate thoughtful comments by Professor Hanan Samet in the early version of my dissertation.

My graduate life in Maryland was full of happy moments thanks to wonderful people around me. In particular, I will never forget the time with Sukhyun Song, Sungwoo Park, Joonghoon Lee, and Hyejung Lee when we settled in Maryland. I am also grateful to my friends and colleague - Suzie Kim, Ginnah Lee, Youngkyung Cho, Jaehwan Lee, Minkyoung Cho, Jisun Shin, Hyungyoung Song, Uran Oh, Takyeon Lee, Hyunjong Cho, Mikyung Kang, Jessica Chang, and Alice Arai.

Lastly, my deepest thanks go to my family - my mother, brothers, husband, and son for their love, support and encouragement. Words cannot express how much I love them and this dissertation is dedicated to them.

Table of Contents

List of Figures vi

1 Introduction 1
1.1 Dynamic Data Structures for Multidimensional Point Sets 4
1.1.1 The Quadtreap: A Randomized Dynamic Data Structure for Approximate Range Searching 7
1.1.2 The Splay Quadtree: A Self-adjusting Data Structure for Multidimensional Point Sets 10
1.2 Output-Sensitive Well-Separated Pair Decompositions for Dynamic Point Sets 13
1.3 Maintaining Nets and Net Trees under Incremental Motion 17
1.4 Organization of the Dissertation 21

2 Literature Review 22
2.1 Dynamic 1-dimensional Data Structures 22
2.1.1 Amortized Efficiency 23
2.1.2 Randomized Dictionary Structures 25
2.1.3 Other Methods for Dynamic Dictionaries 27
2.1.4 Dynamic Data Structures for Maintaining Trees 29
2.2 Hierarchical Spatial Decomposition for Proximity Searching 32
2.2.1 Quadtrees and Variants 33
2.2.2 Quadtrees and Variants 34
2.2.3 Balanced Box Decomposition Tree 36
2.2.4 Balanced Aspect-ratio Tree 39
2.2.5 Other Approaches for Approximate Nearest Neighbor Searching 40
2.3 Well-Separated Pair Decompositions and Spanners 42
2.3.1 Well-Separated Pair Decompositions 43
2.3.2 Spanners 44
2.3.3 Generalization to Metric Spaces 47

3 A Dynamic Data Structure for Approximate Range Searching 52
3.1 Introduction 52
3.2 Incremental Construction of a BD-Tree 55
3.2.1 Incremental Construction Algorithm 58
3.2.2 Node Labels and the Heap Property 62
3.2.3 Randomized Incremental Construction 67
3.3 The Quadtreap 71
3.3.1 Pseudo-nodes and Rotation 71
3.3.2 Point Insertion 74
3.3.3 Point Deletion 80
3.4 Approximate Range Queries 86

4 A Self-adjusting Data Structure for Multidimensional Point Sets 98
4.1 Introduction . . . 98
4.2 Preliminaries . . . 99
4.3 Self-adjusting Quadtrees . . . 103
4.3.1 Basic Splaying Steps . . . 103
4.3.2 Splaying Operation . . . 109
4.4 Insertion and Deletion . . . 117
4.5 Search and Optimality Results . . . 122
4.5.1 Basic Optimality Results . . . 124
4.5.2 The Static Finger Results . . . 126

5 Output-Sensitive Well-Separated Pair Decompositions for Dynamic Point Sets 139
5.1 Introduction . . . 139
5.2 Preliminaries . . . 140
5.2.1 Strong Separation . . . 142
5.3 Construction and Utilities . . . 148
5.4 Updates . . . 153
5.4.1 Insertion . . . 154
5.4.2 Deletion . . . 158
5.4.3 Maintenance of WSPD for updates . . . 159

6 Maintaining Nets and Net Trees under Incremental Motion 165
6.1 Introduction . . . 165
6.2 Preliminaries . . . 167
6.3 Incremental Maintenance of a Slack Net . . . 171
6.3.1 Online Algorithm for Maintaining a Slack Net . . . 172
6.3.2 Competitive Analysis for Maintaining Nets . . . 174
6.4 Incremental Maintenance Algorithm for Net Tree . . . 183
6.4.1 Incremental Maintenance Algorithm for Net Tree . . . 184
6.4.2 Competitive Analysis for Net Tree . . . 187

7 Conclusions 193

Bibliography 199

List of Figures

1.1 Approximate range searching. 8
1.2 Well separated pairs. 14

2.1 A skip list 26
2.2 Link-cut tree 31
2.3 Topology trees: (a) a multilevel partition of the vertices, (b) the corresponding topology tree. 32
2.4 Centroid shrink 38

3.1 Splitting (left) and shrinking (right). In each case the initial cell contains an inner box, which is indicated by a small gray square. . . . 57
3.2 Insertion algorithm 59
3.3 The tree on the left has been created with the sequence $\langle a, b, c\rangle$ and the one on the right with $\langle a, c, b\rangle$. Uppercase letters identify the outer cells of shrink nodes. 60
3.4 Node labeling for the insertion order $\langle a, b, c\rangle$. 63
3.5 The pseudo-node corresponding to a shrink-split pair. 72
3.6 Rotation operation on pseudo-nodes (a). The associated subdivision is shown in (b). (The order of left-right children is based on the assumption that the earliest insertion time (or inner box, if present) lies within the cell associated with subtree A.) 73
3.7 Example of quadtreap restructuring after insertion. 76
3.8 Proof of Lemma 3.3.1. 77
3.9 Example of deletion from a quadtreap. 81
3.10 Illustrating the execution of the inner loop of Algorithm 2. 83
3.11 A link. 88
3.12 A shrink chain and the transformed tree. 90
3.13 Redefined tree primitives. 91

4.1 (a) Left and outer promotions and (b) right promotion. 101
4.2 The basic splaying operations: (a) *zig*, (b) *zig-zag*, and (c) *zig-zig*, where (before promotion) $y = p(x)$ and $z = g(x)$. 104
4.3 Applying simple splaying along a path with $seq_T(u) = OOOLOLL$. . 110
4.4 A *zig-zig* splaying when $str(x) = OR$. 113
4.5 Point insertion. 118
4.6 Illustrating Steps 2–5 of Algorithm 7. 121
4.7 Illustrating Steps 6–9 of Algorithm 7. 121
4.8 (a) Working set, (b) the leaf cells overlapping a ball 132

5.1 Homogeneous separation. 143
5.2 Find the maximum separation level of homogeneous 1-well separated pair 145
5.3 Inserting a point into the compressed quadtree. 154

5.4 A well-separated pair of new leaf x 158

Chapter 1

Introduction

Recent decades have witnessed the development of many data structures for efficient geometric search and retrieval. A fundamental issue in the design of these data structures is the balance between efficiency, generality, and simplicity. From the perspective of computational geometry, one of the greatest successes has been the development of simple, hierarchical partition-based data structures (such as quadtrees, kd-trees, and variants) for solving approximate geometric retrieval problems involving points sets in spaces of relatively low dimension.

In this dissertation, our focus will be on the design of such data structures in a dynamic setting, that is, where points may be inserted, deleted, or may move over time. We are particularly interested in geometric queries that either involve proximity (such as finding the nearest neighbor to a given query point) or involve range searching (such as reporting or counting the subset of points that lie within a given query region). There are many different ways in which to analyze point sets in a dynamic context. Here are some examples:

Worst-case analysis: The most standard form of analysis is to consider the performance of the data structure where points may be inserted or deleted. The principal design issues are how much space is needed, how fast can points be inserted or deleted, and how fast can queries be answered.

In the field of computational geometry, it is common to analyze these quantities from the perspective of worst-case asymptotic complexity. Ideally, in order to store n points, such a data structure requires space $O(n)$, can process insertions and deletions in time $O(\log n)$, and can respond to queries in time $O(\log n + k)$, where k is the size of the query's output. When solving problems approximately, it is common to have additional terms (typically in the space or query time) that depend on the allowed approximation error ε.

Expected-case and amortized analysis: Worst-case analysis may not be the most relevant form of analysis in some applications. For example, when data access patterns are known to be spatially skewed (that is, where some regions of space are much more likely to be queried than others) it may be more meaningful to define a probability distribution over the set of possible queries. The expected case query time for a set of size n is defined as the expected value of the query time, over all possible queries made to this set.

Such an approach makes the strong assumption that the query distribution is known in advance. Often this is not the case. When the query distribution is not known, a good alternative is a self-adjusting data structure. Such a data structure incrementally reorganizes itself in order to optimize its performance for whatever query distribution is presented. When dealing with self-adjusting data structures, it is common to use an amortized analysis. In an amortized analysis, the running times are averaged over a sequence of accesses. This allows occasional inefficient accesses to be balanced against a higher number

of more efficient accesses.

Kinetic data: In the above cases we have considered dynamics to mean the insertion and deletion of points. In many applications, however, dynamics refers to point sets that move over time. In the field of computational geometry, there are two common models used for processing kinetic data sets. First, motion may be highly predictable and follow specified "flight plans." In this model, the motion of individual points is typically given as an algebraic function of time. (This is the model used in the area of kinetic data structures (KDS) [59].) In the second model, points move incrementally and unpredictably over time. In this model, it is common to provide an oracle that reveals the location of a given point at the current time. This is often called the black-box model [38, 52, 79, 114].)

In this dissertation we will explore a number of efficient data structures for geometric point sets in dynamic contexts. First, we will introduce two new quadtree-based data structures, called the *quadtreap* and the *splay quadtree.* The quadtreap is a randomized variant of a quadtree that supports insertion and deletion and has logarithmic height with high probability. The splay quadtree is also a quadtree variant, but this data structure is self-adjusting and achieves efficiency in the amortized sense. Next, we will present an algorithm that dynamically maintains a fundamental data structure in computational geometry, called the *well-separated pair decomposition* (see definitions below). Our approach allows for efficient insertion and deletion of points into the structure. Its principal feature is that the time to insert and delete points is *output sensitive*, meaning that it depends on the number

of structural changes to the decomposition. Finally, we will present an approach to processing points in a kinetic context in the black-box model. We will show how to efficiently maintain a hierarchical structure called a *net tree* (see Section 1.3 for definitions). This structure can be viewed as a generalization of a quadtree in the context of metric spaces. In the rest of this chapter, we will describe these results in greater detail.

1.1 Dynamic Data Structures for Multidimensional Point Sets

Binary search trees are widely used for storing data sets from a totally ordered domain. Efficient worst-case access requires that tree be balanced, meaning its height is $O(\log n)$. As points are inserted or deleted, however balance may degrade, and many different methods have been proposed for restoring balance. Binary search trees can be incrementally restructured through an operation called a *rotation*, which exchanges the positions of two neighboring nodes in the tree. These are used, for example, in AVL trees [2], red-black trees [60], the treap data structure [97], and splay trees [99]. The treap achieves $O(\log n)$ time per operation with high probability, and the splay tree is efficient in the amortized sense, meaning that a sequence of m operations can be performed in $O(m \log n)$ total time. An alternative data structure for ordered dictionaries, which is not based on binary search trees, is the skip list data structure [90], which combines random sampling with a one-dimensional linked list.

A natural generalization of binary trees for storing points in multidimensional

space are partition trees, which are based on hierarchical subdivisions of space (see, e.g., [113]). An well known example is the PR-quadtree. The PR-quadtree is based on a hierarchical subdivision of space into hypercube-shaped cells through a process of repeated bisection [94]. (For the simplicity, we use the term "quadtree" throughout this dissertation to mean "PR-quadtree.") The simplicity and general utility of quadtree-based data structures is evident by the wide variety of problems to which they have been applied, both in theory and practice (see, e.g., [94]). One nice feature of the quadtree, which makes it appropriate for approximate retrieval problems, is that it decomposes space into disjoint hypercubes, each of which is of constant combinatorial complexity and bounded aspect ratio, two important properties in approximate geometric search and retrieval.

Because the size and depth of a quadtree generally depends on the data set's aspect ratio (the ratio of the maximum to minimum inter-point distances), it is common to use compressed quadtrees [20, 30, 63], which (through the elimination of trivial chains) can store n points in space $O(n)$, irrespective of aspect ratio. Since a compressed quadtree may have height $\Theta(n)$, it is often combined with an auxiliary search structure for maintaining its balance. This can be done by applying some form of centroid decomposition to the tree [35, 63] or by imposing a general-purpose data structure, like a link-cut tree [21, 98]. These approaches are rather inelegant, however, due to their reliance on relatively heavyweight auxiliary structures.

A much simpler and more elegant solution, called the *skip quadtree*, was proposed by Eppstein *et al.* [45]. It consists of a series of compressed quadtrees, each for a successively sparser random sample of the point set, together with links between

corresponding nodes of consecutive trees. The result is a multidimensional analog of the well known skip list data structure [90].

Another creative solution is based on reducing the problem into a 1-dimensional domain by sorting the points according to their relative positions along a space filling curve. For example, this can be performed by interleaving the bits of the coordinates of the points, which is sometimes called the Morton order or Z order. The resulting data set can be stored in a 1-dimensional dynamic data structure. Gargantini proposed such a structure, called the *linear quadtree* [53]. Chan [26, 27] presented a simple and clean implementation of this idea, and he showed how to use it to answer approximate nearest neighbor queries efficiently.

Yet another approach for achieving balance is to modify the definition of the decomposition process. Examples of this include the BBD-tree [10, 12, 13], which uses an operation called centroid-shrinking to produce a balanced tree, and the BAR tree [43], which uses splitting planes that are not axis-aligned. (Many of these data structures will be discussed in greater detail in Chapter 2.)

It is noteworthy that, while almost all the 1-dimensional search structures based on binary trees make use of rotations to maintain balance, almost none of the aforementioned multidimensional data structures makes use of local restructuring in the same simple manner. This suggests the question of whether there even exists a simple way to locally restructure quadtree-based data structures. We show that, if one is willing to modify the manner in which quadtrees are decomposed, it is possible to define such a local restructuring operation. We introduce two variants of the quadtree that support a local restructuring operation called *promotion*. Based on

promotion, we propose two dynamic versions of quadtrees. The first data structure, called a *quadtreap*, is randomized, but its worst-case efficiency can be established with high probability. The second data structure, called a *splay quadtree*, is a self-adjusting structure that is efficient in the amortized sense. We discuss our results regarding these structures in greater detail in Section 1.1.1 and 1.1.2.

1.1.1 The Quadtreap: A Randomized Dynamic Data Structure for Approximate Range Searching

An important class of problems for point sets in real multidimensional space is that of storing these points in order to answer range queries efficiently. Consider a set P of n points in $\mathbb{R}^d$, for some constant d. It is common to assume that each point $p \in P$ is associated with a numeric weight. After preprocessing, we are given a query range Q, and the problem is to compute the sum of weights of the points lying within Q. It is generally assumed that the weights are drawn from a commutative semigroup. In some cases the semigroup is actually a group, and if so, subtraction of weights is also allowed. In *approximate range searching*, the range Q is assumed to be approximated by two convex shapes, an inner range Q^- and an outer range Q^+, where $Q^- \subset Q^+$ (see Fig. 1.1). Points lying within the inner range must be counted and points lying outside the outer range must not. It is assumed that the boundaries of these two shapes are separated by a distance of at least $\varepsilon \cdot \mathrm{diam}(Q)$. Space and query times are expressed as a function of both ε and n. The only other assumption imposed on the query shapes is the *unit-cost assumption*, which states

that it is possible to determine whether an axis-aligned hyperrectangle lies entirely inside Q^+ or entirely outside Q^- in constant time.

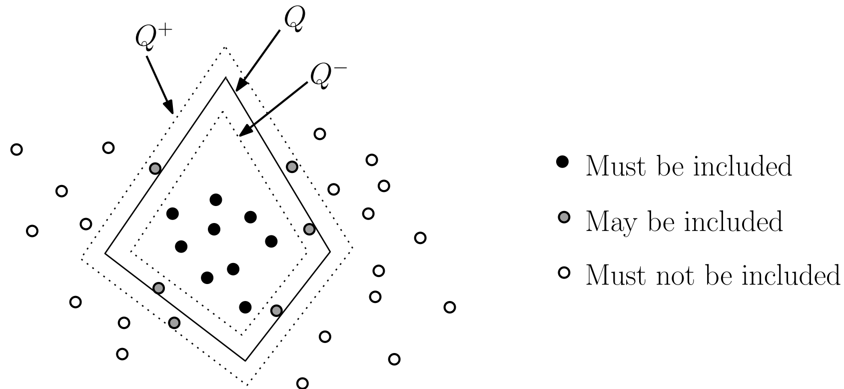


Fig. 1.1: Approximate range searching.

Arya and Mount [12] showed that, under the unit-cost assumption, approximate range queries could be answered with space $O(n)$ and query time $O\big(\log n + (\frac{1}{\varepsilon})^{d-1}\big)$. They showed that this is essentially optimal for methods based on partition trees. Although data structures exist with asymptotically better performance, these methods require that the value of ε be fixed at construction time. The BBD-tree has the nice feature that preprocessing is independent of ε, and hence a single structure can be used to answer queries of all degrees of accuracy.

Efficient data structures have been developed for geometric search and retrieval including approximate range queries for point sets in multidimensional space. Examples include the BBD-tree [10, 12, 13], the BAR tree [43], Chan's linear data structure [26, 27], and the skip quadtree [45]. One significant shortcoming of those data structures, is that none of them admits an efficient solution to the important

problem of approximate range counting together with efficient insertion and deletion. The BBD-tree and BAR tree do not support efficient insertion and deletion (except in the amortized sense, through the process of completely rebuilding unbalanced subtrees [6, 42]). Chan's linear data structure and the skip quadtree both support efficient insertion and deletion, but neither supports range-counting queries. Intuitively, to support range-counting queries, each internal node of a partition tree maintains the total weight of all the points residing in the subtree rooted at this node. With each insertion or deletion, these counts are updated from the point of insertion to the tree's root. In Chan's structure there is no tree, and hence no internal nodes. (It is possible to augment Chan's data structure to impose a tree, but range counting has not been studied for such a data structure.) The skip quadtree is based on the compressed quadtree, which may have height $\Theta(n)$. (The skip quadtree does support efficient range-reporting queries, however, because even an unbalanced subtree of size k can be traversed in $O(k)$ time.)

In Chapter 3, we introduce a simple dynamic data structure, which supports efficient approximate range counting. This tree structure is similar in spirit to the BBD-tree, but whereas the BBD-tree is static, our data structure employs a dynamic rebalancing method based on rotations. Similar to the skip quadtree, which is based on combining quadtrees and the 1-dimensional skip list, our data structure can be viewed as a combination of the quadtree and the treap data structure of Seidel and Aragon [97].

The *treap* is a randomized data structure for storing 1-dimensional keys, which is based on a combination of a binary search tree and a heap. A treap assigns random

priorities to the keys, and it behaves at all times as if the points had been inserted in order of increasing priority. It relies on the fact that, if nodes are inserted in random order into an (unbalanced) binary tree, the tree's height is $O(\log n)$ with high probability. Our structure also assigns priorities to points, and maintains the tree as if the points had been inserted in this order. The tree structure maintains something akin to the heap property of the treap (but each node of our structure is associated with two priority values, not one). Because of its similarity to the treap, we call this data structure a *quadtreap*.

We will show that in any fixed dimension d, it is possible to store a set of n points in a quadtreap of space $O(n)$. The height h of the tree is $O(\log n)$ with high probability. It supports point insertion in time $O(h)$. It supports point deletion in worst-case time $O(h^2)$ and expected-case time $O(h)$, averaged over the points of the tree. It can answer ε-approximate spherical range counting queries over groups and approximate nearest neighbor queries in time $O\big(h + (\frac{1}{\varepsilon})^{d-1}\big)$.

1.1.2 The Splay Quadtree: A Self-adjusting Data Structure for Multidimensional Point Sets

Balanced trees, such as AVL trees, treap, and skip list in one dimensional space, and the skip quadtree, Chan's linear data structures, and our quadtreap data structures in multidimensional space guarantee $O(\log n)$ time (possibly randomized) to access each node of the tree, and so are efficient in the worst case. Since access patterns may be highly skewed, however, it is of interest to know whether the data structure

can achieve even higher degrees of efficiency by exploiting the fact that some nodes are much more likely to be accessed than others. A natural standard for efficiency in such cases is based on the entropy of the access distribution. Entropy-optimal data structures for point location have been proposed by Arya *et al.* [7,9] and Iacono [69]. These data structures are static, however, and cannot readily be generalized to other types of search problems.

A commonly studied approach for adaptive efficiency for one-dimensional data sets involves the notion of a *self-adjusting data structure*, that is, a data structure that dynamically reorganizes itself to fit the pattern of data accesses. The best-known example is the splay tree of Sleator and Tarjan [99, 106]. This is a binary search tree that stores no internal balance information, but nonetheless provides efficient dynamic dictionary operations (search, insert, delete) with respect to *amortized time*, that is, the average time per operation over a sequence of operations. Sleator and Tarjan demonstrated that (or in some cases conjectured that) the splay tree adapts well to skewed access patterns. For example, rather than relating access time to the logarithm of the number of points in the set, it is related to a parameter that characterizes the degree of locality present in a series of access queries. Examples of measures of locality include the working set bound [99], static and dynamic finger bounds [32, 34, 99], the unified bound [22, 67, 99], key-independent optimality [70], and the BST model [40]. Numerous papers have been devoted to the study of dynamic efficiency of data structures (see, e.g., [41, 44, 54, 89, 104, 112]).

Given the fundamental importance of the splay-tree to the storage of one-dimensional point sets and the numerous works it has inspired, it is remarkable

that, after decades of study, there are no comparable self-adjusting data structures for storing multidimensional point sets. In Chapter 4, we propose such a data structure, called the *splay quadtree*. Like the splay tree, the splay quadtree achieves adaptability through a restructuring operation that moves an arbitrary internal node to the root of the tree. (In our tree, data is stored exclusively at the leaf nodes, and so it is not meaningful to bring a leaf node to the root.) Each node of our tree is associated with a region of space, called a *cell*, which is either a quadtree box or the set-theoretic difference of two quadtree boxes, called the inner box and outer box. As seen in other variants of the quadtree, such as the BBD tree [12, 13] and quadtreap [80], this generalization makes it possible to store an arbitrary set of n points in a tree of linear size and logarithmic height.

As with the quadtreap [80], the tree is maintained by a local operation, called *promotion*, which is analogous to rotation in binary trees. The requirement that the cell associated with each node of the tree has at most one inner box limits the set of allowable promotions. Due to this added complication, we cannot directly employ the splaying operation described by Sleator and Tarjan. Instead, our splaying operation involves making three passes along the path from the node being splayed to the root. Nonetheless, we show that the fundamental properties of the standard splay tree apply to our data structure as well.

1.2 Output-Sensitive Well-Separated Pair Decompositions for Dynamic Point Sets

The well-separated pair decomposition (WSPD) is a fundamental structure in computational geometry, which has found numerous applications in the design of geometric approximation algorithms. A set P of n points determines $\Theta(n^2)$ pairs of points. The WSPD provides a concise representation of these pairs using only $O(n)$ space, subject to a desired approximation factor in the accuracy to which distances are represented. In Chapter 5, we present an efficient algorithm for maintaining the WSPD of a dynamic point set, where insertions and deletions are possible.

The concept of the WSPD was introduced by Callahan and Kosaraju [25], but it was anticipated in earlier work by Greengard and Rokhlin on the fast multipole method [56]. Given a positive parameter s, called the *separation factor*, we say that two sets A and B in a metric space are *s-well separated* if they can each be enclosed within balls of equal radii such that the closest distance between these balls is at least s times their common radius (see Fig. 1.2). Given an n-element point set P and $s > 0$, a *well-separated pair decomposition* of P with respect to s (an *s-WSPD*) is a collection of pairs $\{A_1, B_1\}, \ldots, \{A_k, B_k\}$ of non-empty subsets of P such that

(1) for $1 \leq i \leq k$, A_i and B_i are s-well separated, and

(2) for any two distinct points $p, q \in P$, there exists exactly one pair $\{A_i, B_i\}$ such that p lies in one of these sets and q lies in the other.

Throughout, we assume that the metric space is $\mathbb{R}^d$, for a fixed constant d under

the L_∞ distance function. (Later, we will show that the choice of Minkowski metric only affects the constant factors. Well-separated pair decompositions can be readily extended to any metric space of constant doubling-dimension [64].) We treat s and n as asymptotic quantities, and we will assume that $s \geq 1$.

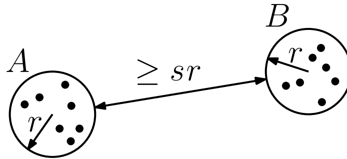


Fig. 1.2: Well separated pairs.

WSPDs have numerous applications in the design of both exact and approximate algorithms for spatial data sets. Given an n-element point set P in $\mathbb{R}^d$, it is possible to use WSPDs to compute the (exact) closest pair of points of P in $O(n \log n)$ time, and generally for $1 \leq k \leq \binom{n}{2}$, it is possible to enumerate the (exact) k closest pairs of points $O(k + n \log n)$ time [100]. It is also possible to compute the nearest neighbor for every point in the set in $O(n \log n)$ time [100]. WSPDs are also used in geometric approximations. For example, it is possible compute an ε-approximation to the diameter of a point set (and generally to approximate the kth smallest interpoint distance) in $O(n \log n + (1/\varepsilon)^d)$ time. It is also possible to use WSPDs for computing approximations to the Euclidean minimum spanning tree [101], constructing geometric spanner graphs (sparse networks whose shortest paths are not much longer than the straight-line distance between points) [81], and building distance oracles for spatial networks [96]. In order to apply any of these algorithms a dynamic context, it is natural to ask how quickly can a WSPD be updated when points are inserted or deleted.

It is well known (see, e.g., [25,63,101]) that, given an n-element point set in $\mathbb{R}^d$, its possible to build an s-WSPD of size $k = O(s^d n)$ in time $O(n \log n + k)$. This is optimal in the worst case, since it is easy to prove that a uniformly distributed point set is expected to require $\Omega(s^d n)$ pairs. It is conventional wisdom, however, that typical point sets are far from uniform. Effects such as clustering, correlations among coordinates, and boundary effects tend to reduce the number of well-separated pairs. This reduction can be quite significant, even in spaces of moderate dimension [82]. We have observed in our own experience with applications in clustering large data sets arising from terrain maps that the number of pairs resulting from the insertion of a single point can vary by two orders of magnitude. For this reason, it is important to consider *output-sensitive algorithms*, that is, algorithms whose running time depends on the number of changes to the WSPD structure. While the aforementioned batch WSPD construction is output sensitive, existing algorithms for maintaining a WSPD are not.

WSPD constructions in $\mathbb{R}^d$ are based on hierarchical spatial subdivisions, and in particular, variants quadtrees and kd-trees. Callahan and Kosaraju used a fair-split tree. Fischer and Har-Peled [48] point out that the construction can be based on the simpler compressed quadtree, which is the approach that we take here (see Section 5.2 for definitions). Construction algorithms follow the same general pattern. The points are stored in the leaves of an appropriate partition tree of size $O(n)$. The k sets of the WSPD are generated by a recursive top-down algorithm and are represented as pairs of nodes of the tree, $\{u_i, v_i\}$ for $1 \leq i \leq k$. The corresponding pairs $\{A_i, B_i\}$ of the decomposition consist of the points of the subtrees rooted at u_i

and v_i, respectively. Construction algorithms generate only *maximal* pairs, meaning that the parent pair, $\{\text{par}(u_i), \text{par}(v_i)\}$, is not s-well-separated.

Clearly, a given point set does not have a unique s-WSPD. Fischer and Har-Peled [48] observed this issue in their algorithm for maintaining the WSPD of a dynamic point set. They observed that the number of well separated pairs can vary based on the choice of the coordinate system's origin, which in turn affects the quadtree structure. They showed this instability could be ameliorated in expectation by applying a random translation to the point set. In particular, they showed that it is possible to maintain an s-WSPD of a point set under insertions and deletions in time $O(s^d(\log n + \log s) \log n)$ with high probability. WSPDs have also been considered in a dynamic context mostly in the context of metric spaces. Examples include the deformable spanner of Gao, Guibas, and Nguyen [52] and dynamic spanners by Gottlieb and Roditty [55] and Roditty [92].

Since the output size is not uniquely determined by the input, this raises the question of what is meant by an "output sensitive" algorithm. Our approach will be to fix a precise notion of s-WSPD procedurally, that is, as the output of a particular algorithm. In order to achieve the best running times, we will strengthen the notion of separation to one that is easy to compute in a quadtree setting. We will show that for any given point set, the size of the s-WSPD that results from our definition is at most a constant factor larger than the size of any (standard) s-WSPD for this point set in any Minkowski L_p metric. Our approach can be readily generalized to any quadtree-like subdivision, and it allows for the possibility of variations such as random translations.

We relate the update time to the number of changes made to the WSPD. In our algorithm (and as with Fischer and Har-Peled), there are two ways that well-separated pairs can change. When a point p is inserted, new pairs of the form $\{\{p\}, B_i\}$ can be *created*, and an existing pair $\{A_i, B_i\}$ can be *modified* to become $\{A_i \cup \{p\}, B_i\}$. Deletion is just reverse of this. Because only newly created or deleted pairs affect our quadtree-based representation, our output-sensitive bounds will be based only on their number. The running time is independent of the number of modified pairs. (By the way, it is reasonable to believe that the number of newly created pairs will be significantly smaller than the number of modified pairs, since they tend to arise in a neighborhood close p, whereas modified pairs can span the entire data set.) Of course, depending on the application, the modified pairs might also be of interest. If desired, it is easy to modify our algorithm to report the modified pairs in an output sensitive manner.

In Chapter 5, we show that it is possible to maintain s-WSPD for a dynamic point set that supports point insertion in worse-case time $O(\log n + m)$, and point deletion in amortized time $O(\log n + m)$, where m denotes the number of newly created (resp., deleted) pairs in the case of insertion (resp., deletion).

1.3 Maintaining Nets and Net Trees under Incremental Motion

Dynamic data structures can be also considered in the context of maintaining points in motion. The problem of maintaining discrete geometric structures for points in motion has been well studied over the years. The vast majority of theoretical work

in this area falls under the category of kinetic data structures (KDS) [59]. KDS are based on the assumption that points move continuously over time, where the motion is specified by algebraic functions of time. This makes it possible to predict the time of future events, and so to predict the precise time in the future at which the structure will undergo its next discrete change.

In practice, however, motion is typically presented incrementally over a series of discrete time steps by a *black-box*, that is, a function that specifies the locations of the points at each time step. For example, this black-box function may be the output of a physics integrator, which determines the current positions of the points based on the numerical solution of a system of differential equations [1,78]. Another example arises in the use of Markov-chain Monte-Carlo (MCMC) algorithms such as the Metropolis-Hastings algorithm [28] and related techniques such as simulated annealing [73]. These are applied in solving geometric optimization problems (see, e.g., [66]). A black-box randomly repeatedly perturbs a set of point locations with each incremental step. After each perturbation, the objective function is reevaluated, and the algorithm may either accept or reject the proposed perturbation.

In both of the aforementioned applications, it is desirable to maintain the current point set in a data structure that can be used for evaluating physical quantities needed by the integrator or for evaluating the objective function. The data structure is updated with each new time step. Examples of useful computations include answering nearest neighbor and range queries [76], or maintaining structures such as well-separated pair decompositions [25], geometric spanners [58], or core sets [3].

In Chapter 6, we introduce such a computational model for the online mainte-

nance of geometric structures under incremental black-box motion. Our approach is similar to the observer-tracker model proposed by Yi and Zhang [114] in the context of online tracking, and is similar in spirit to the IM-MP model of Mount *et al.* [79]. Our model involves the interaction of two agents, an *observer* and a *builder*. The observer monitors the motions of the points over time, and the builder is responsible for maintaining the data structure. These two agents communicate through a set of boolean conditions, called *certificates*. The certificates effectively "prove" the correctness of the current structure (exactly as they do in KDS). Based on the initial point positions, the builder constructs the initial structure and the initial certificates, and communicates these certificates to the observer. The observer monitors the point motion and, whenever it detects that a certificate has been violated, it informs the builder as to which certificates have been violated. The builder then queries the new locations of the points, updates the data structure, and informs the observer of any updates to the certificate set. An algorithm for maintaining a data structure in this model is essentially a communication protocol between the observer and the builder. The total computational cost is defined to be the communication complexity between these two agents. One advantage of this model is that it divorces low-level motion issues from the principal algorithmic issues involving the design of the underlying data structure itself.

For the underlying data structure, we consider the maintenance of nets and net trees. Let P denote a finite set of points in some metric space $\mathcal{M}$, which may be either continuous (as in Euclidean space) or discrete. Given $r > 0$, an *r-net* for P is a subset $X \subseteq P$ such that every point of P lies within distance r of some

point X, and no two points of X are closer than r. (We will actually work with a generalization of this definition, which we will present in Section 6.2.) Each point of P can be associated with a covering point of X that lies within distance r, which is called its *representative*. We can easily derive a tree structure, by building a series of nets with exponentially increasing radius values, and associating each point at level $i-1$ with its representative as parent at level i. Like the Vp-tree [115] and Gh-tree [108], the net tree can be viewed as a metric generalization of hierarchical partition trees like quadtrees [94].

The net tree has a number of advantages over coordinate-based decompositions such as quadtrees. The first is that the net tree is intrinsic to the point set, and thus the structure is invariant under rigid motions of the set. This is an important consideration with kinetic point sets. Another advantage is that the net tree can be defined in general metric spaces, because it is defined purely in terms of distances. A number of papers have been written about improvements to and applications of the above net-tree structure in metric spaces of constant doubling dimension. (See, for example [33, 55, 64, 75].) Note that the net tree is a flexible structure in that there may be many possible choices for the points that form the nets at each level of the tree and the assignment of points to parents.

Our main results are efficient online algorithms for maintaining both nets and net trees for a point set undergoing incremental motion. We analyze our algorithms' efficiency by bounding their competitive ratios relative to an optimal algorithm. Assuming that the points are in a space of constant doubling dimension, we achieve a constant factor competitive ratio for the maintenance of a net and competitive

ratio proportional to the square of the tree's height for the net tree.

1.4 Organization of the Dissertation

The remainder of the dissertation is organized as follows. In Chapter 2 we begin with a review of the relevant literature. In Chapters 3 and 4, we introduce the quadtreap data structure and the splay quadtree, respectively, and analyze their efficiency. In Chapter 5, we present our output-sensitive algorithm for maintaining a well-separated pair decomposition under the operations of insertion and deletion. In Chapter 6, we introduce online algorithms for maintaining nets and net trees for points under incremental motion. Finally, Chapter 7 presents concluding remarks and outlines direction for possible future work.

Chapter 2

Literature Review

In this chapter, we present a survey of the background literature that is relevant to this dissertation. Because of our interest in dynamic data structures, we begin by discussing classical dynamic data structures for storing 1-dimensional point sets. We present approaches that are efficient with respect to a number of different criteria, such as worst-case complexity, amortized complexity, and randomized data structures that are efficient with high probability. Next, we discuss a variety of data structures (both static and dynamic) for storing multidimensional point sets. Most of these structures are based on quadtrees and their variants. We also discuss common queries that can be efficiently answered using these structures, such as approximate nearest neighbor queries and approximate range queries. Finally, we discuss well-separated pair decompositions and spanners.

2.1 Dynamic 1-dimensional Data Structures

An *ordered dictionary* is a data structure for storing 1-dimensional data that supports various operations such as insertion, deletion, and searching. These data structures are of fundamental interest in computer science. To achieve good search times against an arbitrary sequence of insertions and/or deletions, a number of different methods are used to maintain the tree's balance. Examples include AVL

trees [2], red-black trees [60], weight-balanced trees [85], B-trees [15], skip lists [90], and treaps [97]. Efficiency is established under various criteria. Some structures achieve $O(\log n)$ time in the worst case, some are probabilistic and achieve $O(\log n)$ time in expectation or with high probability. Some are efficient in the amortized sense, meaning that a sequence of m operations can be performed in $O(m \log n)$ total time.

2.1.1 Amortized Efficiency

Balanced trees, such as AVL trees, red-black trees, B-trees, and their variants achieve a worst-case time bound of $O(\log n)$ per operation on n-node trees. However, in many applications of search trees, not one but a sequence of operations is performed, and the important issue is the total time to execute the sequence, not the individual times of the operations. In such cases, the appropriate measure of efficiency is the amortized time of the operations, where *amortized time* means the average time of an operation in a worst-case sequence of operations starting with an empty structure.

One way to obtain amortized efficiency is to use a *self-adjusting* data structure. Such a structure is allowed to be in an arbitrary state, but during each operation a simple restructuring rule is applied to improve the efficiency of future operations. The *splay tree* by Sleator and Tarjan [99] is a self-adjusting form of binary search tree. Each access for an arbitrary node is based on an operation called *splaying*. This operation brings the node up to the root by a series of double rotations, followed possibly by one single rotation at the end. This operation has the effect of bringing

frequently accessed nodes near to the root of the tree, in short, it adjusts the tree structure according to the pattern of access. In fact, if the access pattern is highly skewed, it can be even more efficient than data structures that are optimal in the worst case. The splay tree does not need any auxiliary information to be stored at each node.

The amortized bounds for splays are established using a potential argument. *Potential* means prepaid work that can be spent on later operations. The potential is a function of the entire data structure. Let Φ_i denote the potential after i operations and c_i actual cost of the ith operation. Then the amortized cost of the ith operation, denoted a_i is defined to be the actual cost plus the increase in potential:

$$a_i = c_i + \Phi_i - \Phi_{i-1}.$$

So the total amortized cost of n operations is the actual total cost plus the total increase in potential:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$

A potential function is valid if $\Phi_i - \Phi_0 \geq 0$ for all i. If the potential function is valid, then the total actual cost of any sequence of operations is always less than the total amortized cost. Thus, if we define a potential function for the data structure that is initially equal to zero and is always nonnegative, we can derive the amortized cost of an operation by adding its actual cost plus the change in potential.

Sleator and Tarjan defined the potential function for the splay tree to be the sum of the ranks of its nodes, where the *rank* of a node v is defined to be $\lceil \log \mathrm{size}(v) \rceil$, where $\mathrm{size}(v)$ is the number of nodes in the subtree rooted at v. By using a telescoping sum, they prove that the amortized cost of a splay is $O(\log n)$. Every search, insertion, and deletion walks down to a node and then splays this node to the root. Thus all operations can be performed in $O(\log n)$ amortized time.

Another approach to define a tree structure that is efficient in an amortized sense is based upon subtree rebuilding. Andersson [6] and Galperin and Rivest [51] independently discovered such a structure. Andersson called it a *general balanced tree*, and Galperin and Rivest called it the *scapegoat tree*. Unlike the splay tree, the scapegoat tree guarantees a logarithmic worst-case bound on a search and does not restructure the tree during searches. This data structure also supports updates in a logarithmic amortized time without the need to store any extra information at a node. It is maintained by the following simple partial rebuilding rule. After a node is inserted, a walk is performed up to the root along the search path. If a node is encountered in this walk whose height exceeds a given constant factor of the number of nodes in its subtree, its subtree is rebuilt to produce a perfectly balanced tree. Deletion of a node is performed by a lazy deletion rule.

2.1.2 Randomized Dictionary Structures

The *skip list* discovered by Pugh [90] is a data structure that generalizes a linked list and employs probabilistic balancing. A skip list can be viewed as a hierarchy

of successively coarser lists constructed over an original sorted list. For each node in the original list, we make a duplicate of the node with probability 1/2. These duplicates are linked to form a list of roughly half the original size. The process is repeated, and these list are stacked up in levels, where each duplicate stores a pointer to indicate its copy in the previous level (see Fig. 2.1). Since each level of the skip lists contains about half the number of nodes as the previous level, the total number of levels is $O(\log n)$ with high probability.

The search algorithm starts from the highest level. Each level is scanned as far as possible without passing the target value, and it then proceeds down to the next lower level. The worst-case search time is proportional to the number of levels. Update operations are also performed in logarithmic time with high probability by a process similar to the search algorithm.

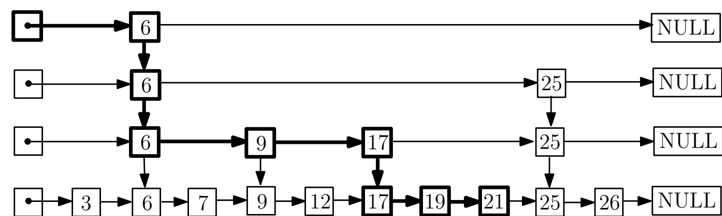


Fig. 2.1: A skip list

A *treap* [1] is a binary tree in which every node has both a key and a priority. The keys are sorted according to an inorder traversal of the tree. The priorities are maintained in heap order, where a parent's priority is smaller than its children. Thus, a treap can be viewed as a binary tree for the keys and a heap for the priorities.

[1] J. Vuillemin introduced the same data structure in 1980 and called it *cartesian tree* [111].

This concept was first introduced by McCreight [77] to describe a related data structure, called a *priority search tree*. This was applied by Seidel and Aragon [97] to define the treap. This data structure assigns a random priority to each newly inserted node. To insert a node, a walk is performed down the treap as in the standard insertion algorithm for binary search trees, and the new node is inserted at the bottom of the tree. In order to reestablish the heap structure, a walk is performed up the treap, and rotations are performed as needed. Deletion is handled by a symmetrical procedure. All operations take time proportional to the depth of the affected node in the tree. Seidel and Aragon prove that the expected depth of any node is $O(\log n)$, which implies that the expected time of these operations matches the worst-case time of the best deterministic tree structures.

2.1.3 Other Methods for Dynamic Dictionaries

If the size of the universe from which keys are derived is sufficiently small, an interesting alternative is the *van Emde Boas tree* [110]. We assume that the keys are drawn from a universe U. Let $u = |U|$. The tree performs the operations, insert, delete, and find in $O(\log \log u)$ time. This bound arises from a binary search on the bits of the key. If a key x has w bits, then we split x into two parts: $high(x)$ and $low(x)$, each with $w/2$ bits. We create $\sqrt{u}$ substructures. Each substructure handles a range of size $\sqrt{u}$ from the universe. These are recursively constructed in the same way. If a key x is stored in one substructure, $S[high(x)]$, then we care only about $low(x)$ in the substructure's children. Each level stores two values, the

minimum element of the set and the maximum element of the set, and the auxiliary tree to keep track of which children are non-empty. This gives a recurrence for the running time of operations of $T(u) = T(\sqrt{u}) + O(1)$, thus, $T(u) = O(\log\log u)$. (The multidimensional variant of this structure is developed by Amir *et al.* [5].)

For some search problems, it may not be possible to develop a data structure that can be incrementally rebalanced to deal with such problems in a dynamic setting. Bently [18] introduced the concept of a decomposable searching problem. A search problem is *decomposable* if the answer for a given input set can be found quickly by combining answers for disjoint subsets of the input. For the decomposable searching problem, Bentley and Saxe [19] show how to build a dynamic system of static structures. This data structure consists of $l = \lfloor \log n \rfloor$ levels, $L_0, L_1, \ldots, L_{l-1}$. Each L_i is either empty or a static data structure storing exactly 2^i items. For any value of n, there is a unique set of levels that must be non-empty. To answer a query, we perform a query on each non-empty level and combine the results. Suppose that there exists a static data structure can store n items after $P(n)$ preprocessing time and answer a query in time $Q(n)$. The query time of the dynamic structure based on such a static structure is at most

$$\sum_{i=0}^{l-1} Q(2^i) < l \cdot Q(n) = O(\log n) \cdot Q(n).$$

If $Q(n) > n^{\varepsilon}$ for any $\varepsilon > 0$, the query time is only $O(Q(n))$. Insertion is similar to the algorithm for incrementing a binary counter, where each level, L_i plays the role of the ith least significant bit. We find the smallest empty level k, build a new data

structure L_k containing the new item and all the items stored in $L_0, L_1, \ldots, L_{k-1}$, and finally discard all the levels smaller than L_k. During the lifetime of the data structure, each item will take part in the construction of $\log n$ different data structures. If we distribute the cost of the total processing time to the nodes involved in the rebuilding process, the total processing time is

$$\sum_{i=0}^{\log n} P(2^i)/2^i = O(\log n) \cdot P(n)/n.$$

If $P(n) > n^{1+\varepsilon}$ for any $\varepsilon > 0$, the amortized time is only $O(P(n)/n)$.

2.1.4 Dynamic Data Structures for Maintaining Trees

Some problems such as network flow and dynamic connectivity require the maintenance of a collection, that is a forest, of vertex-disjoint trees under edge insertion or deletions. The trees can be rooted or free, and vertices and edges may be associated with real-valued costs that may change. The basic query operation is tree membership, that is, while the forest of trees is dynamically changing, which tree contains a given vertex, or whether two vertices are in the same tree.

Sleator and Tarjan [98] introduced the link-cut tree as a structure to maintain a forest of vertex-disjoint rooted trees, each of whose vertices has a real-valued cost. A *link-cut tree* supports the operations, link, cut, and findroot. The *link* operation merges two trees by inserting the edge between the root of a tree and a node in another tree. The *cut* operation deletes the edge between a non-root node and its parent, thus splitting the tree. The *findroot* operation returns the root of the tree

containing a given vertex.

This data structure assumes that the initial forest consists of n single-vertex trees. These trees grow by a link operation and shrink by a cut operation. In order to perform these operations, each tree is partitioned into a collection of vertex-disjoint paths. Each tree operation will be defined in terms of one or more path operations. This partition is defined by classifying each edge as either *solid* or *dashed*. If we consider edges to be directed from child to parent, at most one solid edge enters each vertex. Thus, the solid edges define a collection of *solid paths* that define a partition of the vertices of each tree. When a vertex is accessed, a solid path from this vertex to the root of the tree is formed as follows. Every edge on this path becomes solid, and all the old solid edges that had an endpoint on this path are replaced by dashed edges (see Fig. 2.2). Each tree operation takes time proportional to the length of the solid path. Each solid path defines a sequence of nodes along the path, and this sequence is stored in some type of dynamic tree structure, sorted by the depths of the nodes. The amortized time of $O(\log n)$ per operation is achieved using a dynamic tree [98] or a splay tree [99] as this underlying tree. The worst-case time of $O(\log n)$ is achieved using biased search trees [16].

An alternative for maintaining dynamic trees, called *topology trees*, was introduced by Frederickson [49]. Similar to link-cut trees, topology trees follow the idea of partitioning a tree into a set of vertex-disjoint paths. However, this structure uses a partition based on the topology of the tree.

The basic idea is to partition the tree into a suitable collection of subtrees, called *clusters*, and to implement updates so that only a small number of clusters

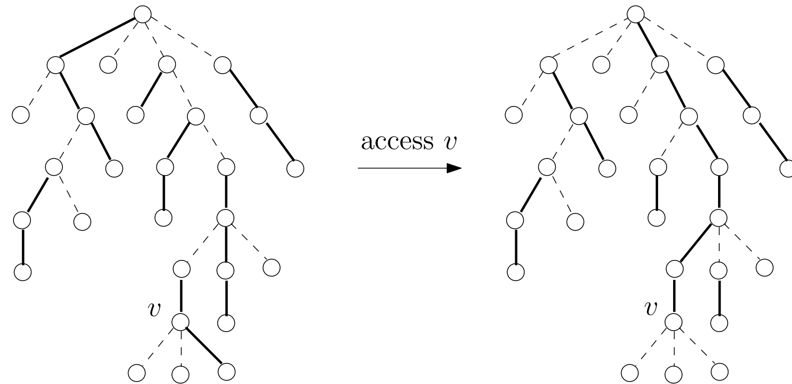


Fig. 2.2: Link-cut tree

are involved with each operation. The decomposition defined by clusters is applied recursively. This structure assumes that no vertex has degree greater than three. (If this is not the case, a well-known transformation in graph theory can be applied [65].)

To define the clustering, let the *external degree* of a cluster be the number of tree edges incident to it. Given a tree of maximum degree 3, a partition is defined such that the following three properties are satisfied:

(i) Each cluster of external degree three has only a single vertex.

(ii) Each cluster of external degree less than three has at most c vertices, for a given constant $c \in \{2, 3, 4\}$.

(iii) No two adjacent clusters can be combined and still satisfy the above properties.

Assuming that a clustering satisfying the above properties can be computed, Frederickson shows how to apply this clustering to form a hierarchical representation of a tree. Clusters at level 0 contain one vertex each. Clusters at level $l \geq 1$ form a partition of the vertices of the tree obtained by shrinking each cluster at level $l - 1$ into a single vertex. (Fig. 2.3 shows the partition of the vertices of a tree and the

corresponding topology tree.) Using the fact over half the clusters at level $l-1$ will be of degree less than three, Frederickson proved that the height of a corresponding topology tree is $\Theta(\log n)$.

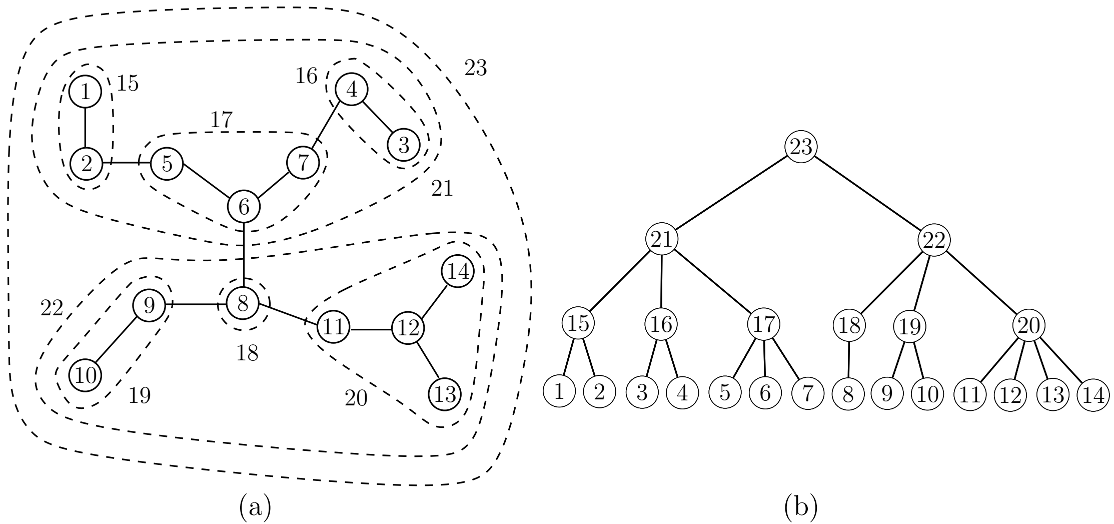


Fig. 2.3: Topology trees: (a) a multilevel partition of the vertices, (b) the corresponding topology tree.

Each level can be updated upon insertion and deletion of any edge by applying a few locally greedy adjustments to the partition. In particular, a constant number of clusters are examined, and these changes percolate up the topology tree, possibly causing other clusters to be regrouped. Thus, update operations can be performed in time proportional to the height of the topology tree, which is $O(\log n)$.

2.2 Hierarchical Spatial Decomposition for Proximity Searching

A number of data structures for multidimensional data have been developed for geometric search and retrieval problems, such as range searching and nearest neighbor

searching. Our focus in this section will be primarily on dynamic data structures for answering such queries for a set of n points in real d-dimensional space. (We will also consider for nondynamic data structures.) Many of these are based on a hierarchical subdivision of space, which gives rise to tree-based structures. That is, they associate a node v in a tree with a region $R(v)$ in d-dimensional space. The children of v are associated with the subregions of $R(v)$ divided by some splitting rule. Different splitting rules define different data structures such as quadtrees [47, 94], k-d trees [17], balanced box decomposition (BBD) trees [12, 13], and balanced aspect-ratio (BAR) trees [42, 43]. Efficiency depends on various properties of the tree, such as its depth and size. One property that is important for many approximate queries is the maximum aspect ratio of any node's region. The *aspect ratio* of a region R in $\mathbb{R}^d$ is defined to be the ratio O_R/I_R, where O_R is the radius of the smallest circumscribed hypersphere in $\mathbb{R}^d$ and I_R is the radius of the largest inscribed d-hypersphere [43].

2.2.1 Quadtrees and Variants

A *PR-quadtree* [94] is a data structure for storing point sets in $\mathbb{R}^d$ based on a hierarchical decomposition of space into axis-aligned hypercubes, which we call *cells*. Starting with an initial hypercube that encloses the point set (which after scaling may be assumed to be the unit hypercube), each cell that contains two or more points is subdivided by splitting it into 2^d identical hyercubes, each of half the side length. This subdivision naturally induces a rooted tree structure, where each internal node

has 2^d children. We refer to the regions resulting from such a process as *quadtree boxes*. (For simplicity, we use the term "quadtree" throughout this dissertation to mean "PR-quadtree".)

Cells have bounded aspect ratio, which is a useful property for achieving good query time. Point insertions and deletions are simple. However, as we have defined it, the tree may have arbitrary depth and size, independent of the number of points. One method for circumventing this problem is path compression [94]. A *compressed quadtree* [20, 30, 63] can store n points using $O(n)$ storage. Regions in this tree are defined in the same way as in a quadtree, but all maximal chains of nodes with only a single non-empty child are compressed, and a single node is associated with the smallest cell containing the data points. Like the quadtree, the resulting decomposition is based on hypercube regions, and so has good aspect ratio. However, a compressed quadtree may have height $\Theta(n)$ for highly skewed point distributions.

2.2.2 Quadtrees and Variants

Eppstein *et al.* [45] introduced an efficient data structure for maintaining a compressed quadtree under insertions and deletions, called a *skip quadtree*. It is similar to a skip list where instead of a linked list we use a quadtree. Given a set P of n points, the skip quadtree is defined as a sequence of compressed quadtrees for samples of P. Given P, consider a sequence $S_0, S_2, \ldots, S_{l-1}$ of subsets of P, such that $S_0 = P$ and S_i is formed by sampling each point of S_{i-1} with probability 1/2. Observe that the number of non-empty subsets, called the *level* l, is $O(\log n)$ with

high probability. For each S_i, we form a compressed quadtree, Q_i for the points of S_i. Every node in Q_i has pointers to the corresponding node in Q_{i-1} and a pointer to the corresponding node in Q_{i+1}, if it exists there.

The skip quadtree provides efficient operations for inserting and deleting points in $O(\log n)$ expected time and answers the following tree types of geometric queries:

- point location search in $O(\log n)$ time,
- $(1+\varepsilon)$-approximate nearest neighbor searching in $O((1/\varepsilon)^{d-1}(\log n+\log(1/\varepsilon)))$ time,
- $(1+\varepsilon)$-approximate range query in $O((1/\varepsilon)^{d-1} + \log n + k)$ time, where k is the size of output.

Let us first explain how searches are performed. To find the smallest quadtree box in Q_0 containing a given query point, q, the search algorithm starts at the topmost quadtree Q_{l-1}. For each tree Q_i visited, it finds the lowest node of Q_i that contains q, and it then moves to the corresponding node in Q_{i-1}, and continues the search from there. Eppstein *et al.* [45] prove that the expected number of nodes examed in each tree is constant. Thus, the search can be performed in $O(\log n)$ expected time.

Insertion of a point p is performed by applying the above search and finding the lowest node v in Q_0 that contains p. Then, it splits v and inserts a new node in Q_0 that contains p. By flipping a coin, it decides whether p belongs to S_1. If not, the process is done. Otherwise, it inserts p to Q_1, and repeats the coin flipping process. Since the amount of work needed at each level is a constant, the total time

is proportional to the number of levels, which is $O(\log n)$. Deletion is similar to insertion.

We will not discuss the other search operations (approximate range query and approximate nearest neighbor searching), but they follow by adapting existing quadtree-based algorithms to this skiplist-like structure.

In addition, the expected space for a randomized skip quadtree is $O(n)$, since with high probability the sizes of the compressed quadtrees in the levels of the skip quadtree forms a geometrical series that converges to $O(n)$.

2.2.3 Balanced Box Decomposition Tree

Arya *et al.* [12, 13] introduced the first tree structure to guarantee both bounded aspect ratio and $O(\log n)$ depth. The structure is a *balanced box-decomposition (BBD) tree*. The BBD tree is a balanced variant of a data structures based on a hierarchical subdivision space into hyperrectangular regions as quadtrees [47, 94], k-d trees [17], or fair-split trees [25, 109]. The principal difference between the BBD tree and these other data structures is that each node of the BBD tree is associated not simply with a d-dimensional rectangle, but with the set theoretic difference of two such rectangles, one enclosed within the other. (In this respect, it is quite similar to binary division trees (BD-trees) [37, 86].)

Every node of a BBD tree is associated with a region of space called a *cell*. A cell is defined to be either an *outer* rectangular box or the set theoretic difference between an *outer* rectangular box and an *inner* rectangular box. Each cell is as-

sociated with the set of data points lying within the cell. The *size* of a cell is the length of the longest side of its outer box. In order to prove bounds in the efficiency of approximate nearest neighbor searching, Arya *et al.* [13] introduced a *stickiness* restriction on the inner box. An inner box is *sticky* if the distance between the inner box and every face of the outer box is either 0 or not less than the size of the inner box. Using stickiness, they proved a packing constraint, which is essential in establishing the efficiency of data structure for geometric queries [12, 13]. This packing constraint states that the number of leaf cells of size at least s that intersect any ball of radius r is at most $O((1 + r/s)^d)$.

The BBD tree is constructed through an alternating application of two operations, *splits* and *shrinks*. They represent two different ways of subdividing a cell into two smaller cells. A (midpoint) split bisects a cell by a hyperplane orthogonal to one of the longest sides. The resulting boxes have aspect ratios of either 1 or 2. A shrink partitions a cell by a box that lies within the original cell. It partitions a cell into two children, one lying inside this box and one lying outside. The shrink operation is actually part of a sequence of up to three operations called a *centroid shrink*. Let n_c denote the number of data points associated with the current cell. The centroid shrink decomposes the current cell into a small number of subcells, each containing at most $2n_c/3$ (see Fig. 2.4). The resulting boxes have bounded aspect ratios.

The centroid shrink operation introduces at most three new levels into the tree, and it is alternated with splits. These follows that with each four levels the number of points decreased by at least a factor of 2/3. Thus, the depth of the BBD

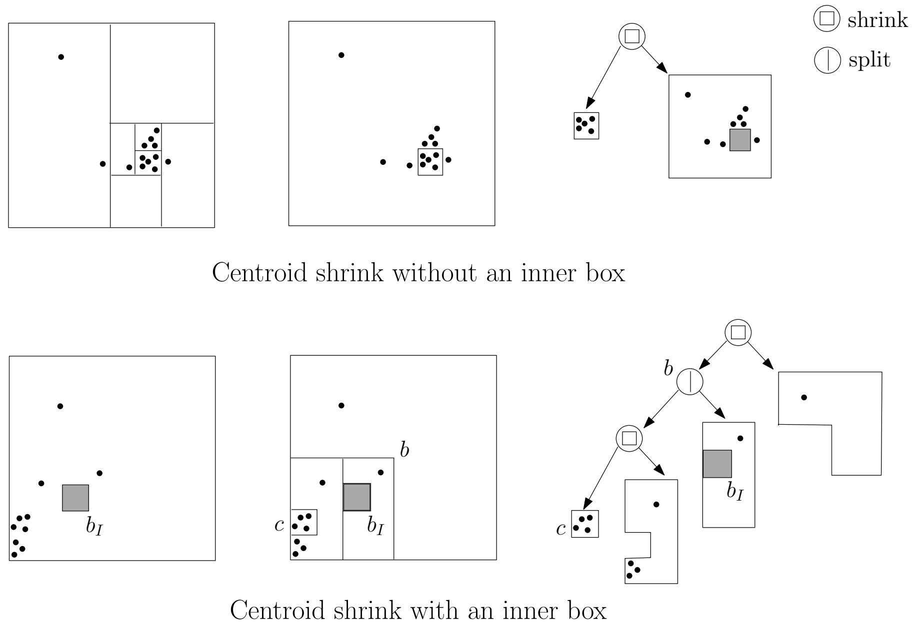


Fig. 2.4: Centroid shrink

tree is $O(\log n)$.

The split operation is preferred because of its simplicity, speed of computation, and the fact that after every d consecutive splits, the geometric size of the associated cell decreases by a constant factor. However, it cannot guarantee that the point set is partitioned evenly. The shrink operation can provide this by rapidly "zooming" into regions where data point are highly clustered. The construction of alternating splits and shrinks guarantees that with every $2d$ levels of descent in the tree, the sizes of the associated cells decrease by at least a factor of 1/2.

By the above size reduction constraint and a packing constraint, it is shown in [13] and [12] that a BBD tree can answer $(1+\varepsilon)$-approximate nearest neighbor queries and $(1+\varepsilon)$-approximate range counting queries in $O((1/\varepsilon)^d \log n)$ time and $O(\log n + (1/\varepsilon)^{d-1})$ time, respectively, with $O(n)$ space.

2.2.4 Balanced Aspect-ratio Tree

The BBD tree partitions space using non-hyperplanar cuts that result in holes, and hence cells are not convex. Several applications in computer graphics and graph drawing require the convexity property of the partitioned regions. (It is possible to further refine these cells into hyperrectangles by the addition of a constant number of hyperplanes. The resulting cells do not necessarily have bounded aspect ratios, but they do satisfy the packing constraint.) A *Balanced Aspect Ratio (BAR) tree* [42,43] simultaneously achieves the desired properties of having $O(\log n)$ depth and regions that are convex and have bounded aspect ratios.

Like the k-d tree, the BAR tree is a binary space partition using hyperplane cuts, but instead of allowing only axis-orthogonal partitions, it also allow a third partition orthogonal to a vector at an angle of $\pi/4$ with respect to the axes, called a *corner cut*. This cut is chosen in such a way that it both divides the point set between the regions evenly, and also ensures that the child regions have bounded aspect ratios. The BAR tree only uses corner cuts when an axis-parallel cut would produce a region does not have bounded aspect ratio.

Duncan *et al.* [43] show that for any $\alpha \geq 3d$ and $\beta \geq d/(d+1)$, their subdivision algorithm produces a sequence of axis-parallel and corner cuts that result in regions of aspect ratio at most α, such that each child has at most a fraction of at most β of the points. By this fact, the depth of a BAR tree is $O(\log n)$.

They show that using $O(n)$ space, $(1+\varepsilon)$-nearest neighbor queries can be answered in $O((1/\varepsilon)^{d-1} \log n)$ time, and $(1+\varepsilon)$-range reporting queries can be answered

in $O(\log n + (1/\varepsilon)^{d-1} + k)$ time, where k is the size of the output.

2.2.5 Other Approaches for Approximate Nearest Neighbor Searching

While the aforementioned data structures are space-efficient, they all have query times of at least $(1/\varepsilon)^{d-1}$, which can be quite high. Har-Peled [62] suggested a faster method for approximate nearest neighbor queries, called the *approximate Voronoi diagram (AVD)*. It is based on the fact that nearest neighbor queries can be reduced to point location in the Voronoi diagram of the point set P. An AVD of P is defined to be a partition of space into cells, where each cell c is associated with a *representative* $r_c \in S$, such that r_c is $(1+\varepsilon)$-approximate nearest neighbor for any query point in c. A cell in the subdivision is a cube or the difference between two axis-aligned cubes, one contained inside the other. These cells are stored in a compressed quadtree, which (with the aid of an appropriate auxiliary structure) can efficiently answer point location queries, thus approximate nearest neighbor queries. For n points in dimension d, this structure requires $O((n/\varepsilon^d)(\log n)(\log(n/\varepsilon)))$ space and answers queries in $O(\log(n/\varepsilon))$ time.

The construction of this data structure can be viewed as a tournament between the sites for "balls of influence", growing around each point. It uses an approximation to the minimum spanning tree to define this tournament.

Arya *et al.* [8] generalized the result of Har-Peled to a structure they called a (t, ε)-*AVD*. It allows up to some given number $t \geq 1$ of representatives to be

associated with each cell c for a given ε, where each point in c has one of these t representatives as its $(1+\varepsilon)$-approximate nearest neighbor. This AVD provides a space-time tradeoff. Arya *et al.* [8] showed that when $t = 1$, this structure requires $O((n/\varepsilon^{d-1})\log(1/\varepsilon))$ space and answers queries in $O(\log(n/\varepsilon))$ time, and when $t = O(1/\varepsilon^{(d-1)/2})$, it requires $O(n\log(1/\varepsilon))$ space and answers queries in $O(t + \log n)$ time. Note that when $t = 1$, this structure is more space efficient than that of Har-Peled by a factor of roughly $O(1/\varepsilon)$. This AVD is based on the BBD tree, and uses a well-separated pair decomposition of the points to define the subdivision process.

Chan [26, 27] introduced one of the simplest implementations of an approximate nearest neighbor data structure. It is an in-place data structure, that is, it requires no extra storage except for the input array, which stores an appropriate permutation of the points. It consists of a sorted array of points, where the ordering will be defined below. If the array is replaced by an ordered dictionary (e.g., skip list or balanced tree) insertions and deletions can be performed in $O(\log n)$ time.

Chan's structure is based on a bit-interleaved ordering of point coordinates, sometimes called the *Morton order* or *Z-order* [94]. Assume that coordinates are positive integers bounded by $U = 2^w$. A point p in d-dimensional space is represented in binary as $(p_{1w}\cdots p_{10}, p_{2w}\cdots p_{20}, \ldots, p_{dw}\cdots p_{d0})$. Its *shuffle* is defined to be the $w \cdot d$ bit binary number formed by concatenating the highest order bits of all the coordinates, followed by the second highest bits, and so on. Thus, the shuffle of p is the binary number $p_{1w}p_{2w}\cdots p_{dw}\cdots p_{10}p_{20}\cdots p_{d0}$. For a number s, the *shifted* point by s is defined to $p + (s, s, \ldots, s)$.

A key property of the Morton order is that all the points that fall within any

quadtree box appear consecutively in the shuffle order. Another important property is that, given two points that are within distance r of each other, with constant probability after a random shift both points lie within a quadtree box of diameter $O(r)$. Based on these observations, Chan shows that $(1+\varepsilon)$-approximate nearest neighbor searching can be performed by first sorting points shifted by a randomly chosen number, and then performing a modified binary search. Chan shows that this can be performed in $O((1/\varepsilon)^d \log(1/\varepsilon) \log n)$ expected time.

2.3 Well-Separated Pair Decompositions and Spanners

A number of problems in computational geometry involve distances determined by pairs of points in a given point set. Given a set P of n points in $\mathbb{R}^d$, where d is a constant, these includes the closest pair of distinct points in P, the largest distance between any two points of P, the nearest neighbor of each point of P, or the minimum spanning tree of P. Most of these problems can be expressed as graph problems on the complete Euclidean graph for P, in which the weight of each edge (p,q) is the Euclidean distance $|pq|$ between p and q. Since the number of edges in this graph is $\Theta(n^2)$, the problems involving pairwise distances can be solved in $\Theta(n^2)$ time. There are approaches that approximately solve them in subquadratic time. Two of these approaches are well-separated pair decompositions (WSPDs) and spanners. We will discuss these two concepts in the next two sections.

2.3.1 Well-Separated Pair Decompositions

A *well-separated pair decomposition (WSPD)* [25] is a partition of the $\binom{n}{2}$ edges of the complete Euclidean graph into $O(n)$ subsets. For any set P of points in $\mathbb{R}^d$, let $R(P)$ denote its *bounding box*. For $s > 0$, point sets A and B are *s-well separated*, if $R(A)$ and $R(B)$ can each be contained in balls of radius r whose minimum distance is at least sr. The real number s is called the *separation factor*. Given arbitrary sets A and B, the *interaction product*, denoted $\otimes$, between A and B is defined to be the set of all distinct (unordered) pairs from these sets, that is,

$$A \otimes B = \{\{a, b\} | a \in A, b \in B, a \neq b\}.$$

Note that $A \otimes A$ is the set of all distinct pairs of A. Given a point set P and a separation factor $s > 0$, an *s-well separated pair decomposition (WSPD)* for P, is a set $\{\{A_1, B_1\}, \{A_2, B_2\}, \ldots, \{A_k, B_k\}\}$, such that

(i) $A_i, B_i \subset P$, for $1 \leq i \leq k$

(ii) $A_i \cap B_i = \emptyset$, for $1 \leq i \leq k$

(iii) $(A_i \otimes B_i) \cup (A_j \otimes B_j) = \emptyset$, for all i, j, such that $1 \leq i < j \leq k$

(iv) $\bigcup_{i=1}^{m} A_i \otimes B_i = P \otimes P$

(v) A_i and B_i are s-sell separated, for $1 \leq i \leq k$.

Observe that a WSPD always exists by setting the $\{A_i, B_i\}$ pairs to each of the distinct pair singletons of P. However, the size of this WSPD is $\binom{n}{2}$. Callahan and Kosaraju [23, 25] show how to construct an s-WSPD of size $O(n)$ using

a data structure called a fair split tree. In fact, their construction works on any hierarchical spatial data structure into cell, of bounded aspect ratio, such as compressed quadtree. They also show that WSPDs can be used to solve a variety of distance-related problems, such as the k closest pair, the all-nearest neighbors, and the approximate minimum spanning tree.

Let T denote the fair split tree for the point set P. Note that every internal node of a fair split tree has two children. For any node u of T, let P_u denote the subset of P that is stored at the leaves of the subtree rooted at u. A WSPD for P is constructed recursively. For each internal node u of T, the construction performs the following and returns the union of the results. Let v and w be the two children of u. If the sets P_u and P_w are well-separated, then return the pair $\{P_u, P_w\}$. Otherwise, assume without loss of generality that the size $R(P_v)$ is not larger than $R(P_w)$. For each child w_i of w and v, it performs a recursive call for w_i and v, and returns the union of the resulting well-separated pairs. Callahan and Kosaraju [23, 25] show that for a given separation factor s, an s-well separated pair decomposition for P, consists of $O(s^d n)$ pairs, and can be computed in $O(n \log n + s^d n)$ time.

2.3.2 Spanners

Another approach to approximating the distance information for a point set is a spanner [81]. Given a set of points P and a parameter $t \geq 1$, called the *stretch factor*, a *t-spanner* is defined to be a weighted graph G whose vertex set is P, and

for any pair of points $p, q \in P$,

$$|pq| \leq \delta_G(p, q) \leq t \cdot |pq|,$$

where $\delta_G(p, q)$ denotes the length of the shortest path between p and q in G.

The measures of a spanner's sparseness include its size, weight, degree, and diameter. The *size* of a spanner is defined to be the number of edges in the graph. The *weight* of a spanner is the sum of its edge weights. The *degree* is the maximum number of edges incident to any vertex. The *diameter* is defined as the smallest integer D such that for any pair p and q of points there is a t-spanner path from p to q containing at most D edges. Good t-spanners require small size, weight, degree and diameter. The size of any spanner must be at least $n - 1$. Because a spanner must be connected, the weight of a spanner must be at least the weight of a minimum spanning tree for P, $wt(MST(P))$. It is possible to achieve trade-offs between the degree, the diameter and the size [102].

Spanners for complete Euclidean graphs as well as for arbitrary weighted graphs can be used to solve or approximate version of various optimization problems such as minimum spanning trees [24], shortest paths [31], traveling salesperson tour [91], and minimum-cost multi-connectivity in geometric networks [36]. They can also be used in areas such as metric space searching [83] and broadcasting in communication networks [46].

The most well-known algorithms for the construction of t-spanners are spanners constructed from WSPDs, greedy spanners, and Θ-graphs. We discuss each of

these below.

The construction of a t-spanner using a WSPD is done by first constructing a WSPD of P with a separation factor $s = 4(t+1)/(t-1)$. Initially the spanner graph G consists of the vertex set P and an empty edge set. Add edges iteratively as follows. For each well-separated pair $\{A, B\}$ in the decomposition, an edge (a, b) is added to the graph, where a and b are arbitrary points in A and B, respectively. The resulting graph is a t-spanner for P with $O(s^d n) = O(n/(t-1)^d)$ edges and can be constructed in time $O(n \log n + s^d n) = O(n(\log n + 1/(t-1)^d))$ [24]. This spanner has an $\Omega(n)$ worst case lower bound on the degree and the diameter. (To see this, consider the case of n points on a line with exponentially decreasing interpoint distance from left to right.)

The greedy algorithm for spanners appears to have been discovered independently by Bern (unpublished) and Althofer *et al.* [4]. The spanners constructed using the greedy algorithm are called *greedy spanners*. The greedy algorithm maintains a partial t-spanner G' while processing all point pairs in order of increasing edge weight. Processing a pair p, q entails a shortest path query on G' between p and q. If the shortest path in G' is at most $t \cdot |pq|$ then the edge (p, q) is discarded, otherwise it is added to G'. This algorithm takes $O(n^3 \log n)$ time using $O(n^2)$ space. Gudmundssson *et al.* [57] show the improved result that a t-spanner of P with $O((n/(t-1)^d) \log(1/(t-1)))$ edges, degree $O((1/(t-1)^d) \log(1/(t-1)))$, weight $O((1/(t-1)^{2d}) \cdot wt(MST(P)))$ can be computed by the greedy method in time $O((n/(t-1)^{2d}) \log n)$.

The Θ-graph was discovered independently by Clarkson [31] and Keil [71].

The Θ-graph algorithm processes each point $p \in P$ as follows. Given a parameter $\Theta > 0$, partition $\mathbb{R}^d$ into k simplicial cones of angular diameter at most θ and apex at p, where $k = O(1/\theta^{d-1})$. For each non-empty cone C, add an edge between p and the point of C whose orthogonal projection onto some fixed ray in C emanating from p is closest to p. The Θ-graph is a t-spanner of P for $t = 1/(\cos\theta - \sin\theta)$ with $O(n/\theta^{d-1})$ edges and can be computed in $O(n/\theta^{d-1}\log^{d-1} n)$ time using $O(n/\theta^{d-1} + n\log^{d-2} n)$ space [72, 93]. Arya *et al.* [14] introduce a *skip list spanner* combining the Θ-graph with skip lists. A skip list spanner supports insertions and deletions in $O(1/\theta^{d-1}\log^d n \log\log n)$ expected amortized time, having $O(n/\theta^{d-1})$ edges and $O(\log n)$ diameter with high probability.

2.3.3 Generalization to Metric Spaces

All results in the previous sections are valid for d-dimensional Euclidean space, where the dimension, d is a constant. The efficiency of most algorithms depends on the assumption of constant dimension. This is because the running times of many algorithms grow exponentially with the dimension, a phenomenon called "the curse of dimensionality". In many applicative the distance metric is not Euclidean, but still behaves in a manner similar to Euclidean space of constant dimension. There are a number of ways to define a notion of dimension on a finite metric space.

One such concept is the notion of doubling dimension [33, 55, 64, 75, 105]. The set of points within distance r of a point is called the *ball* of radius r centered at that point. The *doubling dimension* of a metric space $\mathcal{M}$ is the minimum value

λ such that every ball b in $\mathcal{M}$ can be covered by at most 2^λ balls of half the radius. A metric is *doubling* if its dimension is $O(1)$. The space $\mathbb{R}^d$ with Euclidean metric has doubling dimension $O(d)$, and so doubling dimension can be thought of as a generalization of the standard notion of dimension [61]. Doubling metrics occur naturally in practical applications such as peer-to-peer networks [84] and data analysis (e.g., when the input data resides on a low-dimensional manifold [107]).

Many algorithmic results on doubling metrics are based on the notion of hierarchical nets. Consider a set of points P in some metric space $\mathcal{M}$. For $p, q \in P$, let $d(p, q)$ denote their distance. A subset $X \subseteq P$ is a r-net if it satisfies the following conditions:

- (i) for every pair $p, q \in X$, $d(p, q) \geq r$,
- (ii) every point of P is within distance r of some $p \in X$.

Krauthgamer and Lee [75] introduced hierarchical nets composed of levels of r-nets, called *navigating nets* or *net trees*. Consider the maximum and minimum interpoint distances in P, d_{min} and d_{max}, respectively. Let $\Gamma = \{2^i : \lfloor \log d_{min} \rfloor \leq i \leq \lceil \log d_{max} \rceil\}$. Each value $r \in \Gamma$ is called a *scale*. For every $r \in \Gamma$, the net level X_r is an r-net of the next lower level $X_{r/2}$. The bottom level of the navigating net contains all points of P, and the top level contains a single point. For every $r \in \Gamma$ and every $p \in X_r$, the data structure stores a list of the nearby points to p among the $r/2$-net $X_{r/2}$. This *scale r navigation list of* p is defined by

$$L_{p,r} = \{q \in X_{r/2} : d(q, p) \leq \gamma \cdot r\},$$

where $\gamma > 0$ is a universal constant. For every scale r, the following properties hold:

(*i*) Covering property: for every $q \in P$, $d(q, X_r) < 2r$,

where $d(q, X_r) = \min_{p \in X_r} d(q, p)$.

(*ii*) Packing property: for every $p_1, p_2 \in X_r$, $d(p_1, p_2) \geq r$.

Krauthgamer and Lee [75] applied navigating nets to solve $(1+\varepsilon)$-approximate nearest neighbor queries on a point set. Letting $\Phi = d_{max}/d_{min}$ (called the *spread* of P), the query time is $O(2^{O(\lambda)} \log \Phi + (1/\varepsilon)^{O(\lambda)})$. This data structure has $O(2^{O(\lambda)} n)$ size and supports insertions and deletions in $O(2^{O(\lambda)} \log \Phi \log \log \Phi)$ time.

Cole and Gottlieb [33] removed the dependency in the spread for the approximate nearest neighbor searching. They presented a data structure for storing a set of n points in any metric spaces of constant doubling dimension. This data structure has linear size, supports insertions and deletions in $O(\log n)$ worst case time, and can find a $(1+\varepsilon)$-approximate nearest neighbors in time $O(\log n + (1/\varepsilon)^{O(1)})$. They base their algorithm on a variant of the hierarchical structure of [75].

Another important problem in metric spaces is computing well-separated pair decompositions. Note that the earlier definition in Euclidean space can be generalized naturally to metric spaces. Talwar [105] extended the WSPDs of Callahan and Kosaraju [25] to metric spaces. They showed that for every set P of n points in a metric space of doubling dimension λ and spread Φ, there exists an s-WSPD with $O(s^{O(\lambda)} n \log \Phi)$ pairs. This is weaker than the result of Callahan and Kosaraju [25] for Euclidean space, which does not depend on the spread of the point set. Har-Peled and Mendel [64] improved this result. They showed that an s-WSPD with

$O(s^{O(\lambda)}n)$ pairs can be computed in $O(2^{O(\lambda)}n \log n + s^{O(\lambda)}n)$ expected time.

Even though the net tree data structure was developed for applications in metric spaces, they have been applied to problems in Euclidean space as well. For example, Gao *et al.* [52] presented a dynamic and kinetic spanner in Euclidean space using an hierarchical structure similar to the net tree of Krauthgamer and Lee [75], which they called a *deformable spanner*. For a set of n points P with spread Φ, they employed a hierarchical structure T having $O(\log \Phi)$ levels, where each level i maintains a 2^i-net of the lower level. Each vertex v on level i in T is connected by edges to all other vertices on level i within distance $O(2^i/(t-1))$ of v. They proved that the resulting graph is a t-spanner for P with size $O(n/(t-1)^d)$ and degree $\mathrm{O}(\log \Phi/(t-1)^d)$. They also showed how to perform updates in time $O(\log \Phi/(t-1)^d)$ per insertion and deletion. They generalized deformable spanners to the kinetic case. Kinetic data structures involve storing point sets that move according to known motion plans. These plans are pseudo-algebraic, that is, they are typically assumed to be given by piecewise algebraic curves. They showed that the total number of events in maintaining the deformable spanner with a stretch factor t is $O(n^2 \log \Phi)$ under pseudo-algebraic motion, and for each event, the structure can be updated in $O(\log \Phi/(t-1)^d)$ time.

Gottlieb and Roditity [55] developed a dynamic spanner in metric spaces based on the deformable spanners of Gao *et al.* [52]. The partition by Gao *et al.*has individual points appearing in as many as $O(\log \Phi)$ levels. In order to avoid the dependence on Φ in the update time of the hierarchy and the spanner, they introduce a process of point replacements that limited the number of times a single point

can appear in the hierarchy. The resulting t-spanner has constant degree and can support insertions and deletions in time $O(\log n)$, where the doubling dimension λ and a stretch factor t are taken as constants.

Chapter 3

A Dynamic Data Structure for Approximate Range Searching[1]

3.1 Introduction

Quadtrees and their many variants have proved to be an efficient and flexible method of storing points in real multidimensional spaces of low dimension. (See, e.g., [94] for many examples of such structures.) These data structures can answer various types of queries efficiently, particularly when approximation is involved. The queries of interest to us include approximate nearest neighbor queries and approximate range queries (see Chapter 1 for definitions). In this chapter, we consider the question of how to store dynamic point sets that allow insertion and deletion in order to answer such queries efficiently. Our focus here will be on establishing good asymptotic bounds on the space and query time of the data structure.

Static data structures have been developed for answering nearest neighbor and range search queries approximately. Examples include the BBD-tree [10, 12, 13] and the BAR tree [43]. These data structures can store a set of n points in $\mathbb{R}^d$ in $O(n)$ space and can answer ε-approximate nearest neighbor queries and ε-approximate range queries in $O(\log n + (1/\varepsilon)^{d-1})$ time. However, these data structures do not

[1] The material appearing in the chapter is based on the paper "A Dynamic Data Structure for Approximate Range Searching" [80].

naturally support insertion or deletion of points.

Dynamic data structures have been proposed for these problems. Chan's implementation of the linear quadtree [26, 27], and the skip quadtree of Eppstein, Goodrich, and Sun [45] both support insertion and deletion. Chan shows that the linear quadtree can efficiently answer approximate nearest neighbor queries in $O((1/\varepsilon)^d \log n)$ time. Eppstein, Goodrich, and Sun show that the skip quadtree can answer approximate range reporting queries in time $O((1/\varepsilon)^d \log n + k)$, where k is the number of points reported. One significant shortcoming of both of these dynamic data structures, is that neither of them admits an efficient solution to the important problem of approximate range *counting*. Intuitively, range-counting is harder because it requires internal nodes to efficiently aggregate information from their children. For example, each internal node of a partition tree maintains the total weight of all the points residing in the subtree rooted at this node. In order to answer a query, a minimal set of nodes is identified which together form a disjoint cover of the query region. The weights of these nodes are then summed together. With each insertion or deletion, these node counts need to be updated. In the linear quadtree structure there is no tree, and hence no internal nodes. (It is possible to augment Chan's data structure to impose a tree, but range counting has not been studied for such a data structure.) The skip quadtree is based on the compressed quadtree, which may have height $\Omega(n)$. Consequently, updating the weights of the internal from a leaf to the root can take $\Omega(n)$ time. (The skip quadtree does support efficient range-reporting queries, however, because even an unbalanced subtree of size k can be traversed in $O(k)$ time.)

In this chapter, we introduce a simple, randomized dynamic data structure, called a *quadtreap*, for storing multidimensional point sets, which supports efficient approximate range counting. This data structure is a randomized, balanced variant of a quadtree data structure. It can be viewed as a multidimensional generalization of the treap data structure of Seidel and Aragon [97]. When inserted, points are assigned random priorities, and the tree is restructured through rotations as if the points had been inserted in priority order. Our main result is stated below. Unlike the skip quadtree, the quadtreap (with high probability) is of logarithmic height, and so it is possible update node weights efficiently. Here is the main result of this chapter.

Theorem 3.1.1 *Given a set of n points in $\mathbb{R}^d$, a quadtreap storing these points has space $O(n)$. The tree structure is randomized and, with high probability, it has height $O(\log n)$. Letting h denote the height of the tree:*

(i) It supports point insertion in time $O(h)$.

(ii) It supports point deletion in worst-case time $O(h^2)$, and in expected-case time $O(h)$ (averaged over all the points of the tree).

(iii) Approximate range-counting queries can be answered in time:

- *$O\big(h + (\frac{1}{\varepsilon})^{d-1}\big)$, when the point weights are drawn from a commutative group,*
- *$O\big(h \cdot (\frac{1}{\varepsilon})^{d-1}\big)$, when the point weights are drawn from a commutative semigroup.*

(iv) Approximate range-reporting queries can be answered in time $O\big(h+(\frac{1}{\varepsilon})^{d-1}+k\big)$*, where* k *is the output size.*

For approximate range-counting, it is assumed that the inner and outer ranges are convex, and satisfy the unit-cost assumption. (See the definitions given in Chapter 1).

The rest of the chapter is organized as follows. In Section 3.2, we introduce the BD-tree data structure and present a simple incremental algorithm for its construction. In Section 3.3, we present our dynamic data structure and show how insertions and deletions are performed. In Section 3.4, we present the range query algorithm.

3.2 Incremental Construction of a BD-Tree

To motivate our data structure, we begin by recalling some of the basic elements of quadtrees and BD-trees. A *quadtree* is a hierarchical decomposition of space into d-dimensional hypercubes, called *cells*. The root of the quadtree is associated with a unit hypercube $[0,1]^d$, and we assume that (through an appropriate scaling) all the points fit within this hypercube. Each internal node has 2^d children corresponding to a subdivision of its cell into 2^d disjoint hypercubes, each having half the side length. Given a set P of points, this decomposition process is applied until each cell contains at most one point, and these terminal cells form the leaves of the trees. A *compressed quadtree*, is obtained by replacing all maximal chains of nodes that have a single non-empty child by a single node associated with the coordinates of

the smallest quadtree box containing the data points. The size of a compressed quadtree is $O(n)$, and it can be constructed in time $O(n \log n)$ time (see, e.g., [30]).

If the dimension of the space is much larger than a small constant, it is impractical to split each internal node into 2^d cells, many of which may contain no points. For this reason, we consider a binary variant of the quadtree, which is closely related to bintrees [74,95]. Each decomposition step splits a cell by an axis-orthogonal hyperplane that bisects the cell's longest side. If there are ties, the side with the smallest coordinate index is selected. Each cell has constant aspect ratio, and so the essential packing properties of cells still hold. Henceforth, we use the terms *compressed quadtree* and *quadtree box* in this binary context. As with standard quadtree boxes, it is easy to prove that two quadtree boxes in this binary context are either spatially disjoint or one is nested within the other. Assuming a model of computation that supports exclusive bitwise-or and base-2 logarithms on point coordinates, the smallest quadtree box enclosing a pair of points (or more generally a pair of quadtree boxes) can be computed in constant time.

If the point distribution is highly skewed, a compressed quadtree may have height $\Theta(n)$. One way to deal with this is to introduce a partitioning mechanism that allows the algorithm to "zoom" into regions of dense concentration. In [12,13] a subdivision operation, called *shrinking*, was proposed to achieve this. The resulting data structure is called a *box-decomposition tree* (*BD-tree*[2]). This is a binary tree in which the cell associated with each node is either a quadtree box or the set theoretic

[2]This data structure is closely related to the binary division tree [37, 86], which remarkably shares the same acronym, BD-tree.

difference of two such boxes, one enclosed within the other. Thus, each cell is defined by an *outer box* and an optional *inner box*. The *size* of a cell is defined to be the maximum side length of its outer box. Although cells are not convex, they have bounded aspect ratio and are of constant combinatorial complexity.

We say that a cell of a BD-tree is *crowded* if it contains two or more points or if it contains an inner box and at least one point. Each crowded cell is partitioned into two smaller cells by one of two partitioning operations (see Fig. 3.1). The first operation, called a *split*, partitions a cell by an axis-orthogonal hyperplane in the same manner described above for the binary quadtree. The associated node has two children, one associated with each of these cells, called the *left child* and *right child*. In contrast to traditional partition-tree definitions, we do not assume that the coordinates of points in the left node are smaller than those of the the right node. Instead (for reasons to be discussed later), it will be convenient to assume that, if the original cell contains an inner box, this inner box lies within the left cell. We call this the *inner-left convention*. Otherwise, the nodes may be labeled left and right arbitrarily.

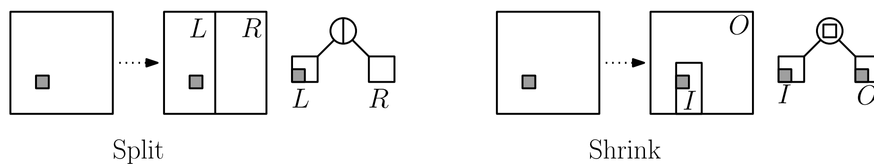


Fig. 3.1: Splitting (left) and shrinking (right). In each case the initial cell contains an inner box, which is indicated by a small gray square.

The second operation, called a *shrink*, partitions a cell by a quadtree box (called the *shrinking box*), which lies within the cell. It partitions the cell into

two parts, one lying within the shrinking box (called the *inner cell*), and the other being the set-theoretic difference of the original cell and the shrinking box (called the *outer cell*). The associated child nodes are called the *inner child* and *outer child*, respectively. (In our figures, the inner child will always be on the left side.)

We will maintain the invariant that, whenever a shrink operation is applied to a cell that contains an inner box, the shrinking box (properly) contains the inner box. This implies that a cell never has more than one inner box. We allow a degenerate shrink to occur, where the shrinking box is equal to the cell's outer box. In this case, the outer box is said to be a *vacuous leaf* since it has no area, and hence no points can be inserted into it. In summary, there are two types of internal nodes in a BD-tree, *split nodes* and *shrink nodes*, depending on the choice of the partitioning operation.

The decomposition process terminates when the cell associated with the current node contains either a single point or a single inner box. The resulting node is a *leaf*.

3.2.1 Incremental Construction Algorithm

In [12, 13], a bottom-up method for constructing a BD-tree of height $O(\log n)$ was proposed. The starting point for our discussion is a simple (top down) incremental algorithm for constructing a BD-tree through repeated insertion. This construction does not guarantee logarithmic height, but in Section 3.2.3 we will show that, if the points are inserted in random order, the height will be $O(\log n)$ with high probability.

Ignoring the trivial case, where the tree is empty, we will maintain the invariant that the cell associated with each leaf node contains either a single point or a single inner box, which we call its *contents*.

Given a point set $P = \{p_1, \ldots, p_n\}$, the incremental construction algorithm begins with an empty tree, whose only cell is the unit hypercube, and it proceeds by repeatedly inserting each point $p \in P$ into the tree as follows. First, through a top-down descent, we find the leaf node u that contains p (see Fig. 3.2). Let b denote u's contents (either a point or an inner box), and let C denote u's outer box. We compute the minimum quadtree box enclosing both p and b, denoted E. By the minimality of E and basic properties of quadtree boxes, applying a split to E will produce two cells, one containing b and the other containing p.

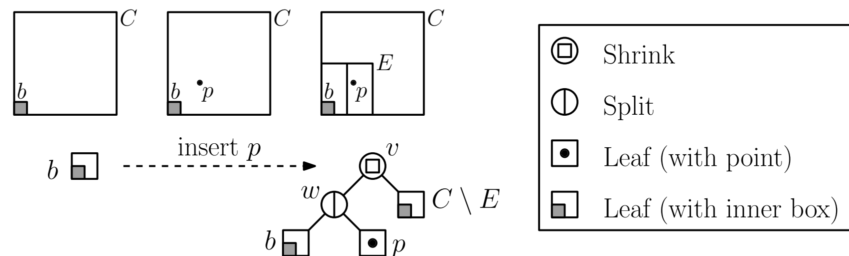


Fig. 3.2: Insertion algorithm

We replace u with a pair of internal nodes v and w. The node v shrinks from u's outer box down to E. The outer child of v is a leaf node with outer box C and inner box E. (Note that E might be equal to u's outer box C), in which case the outer child is a vacuous leaf.) The inner child of v is a split node w that separates b from p. The children of w are two leaves. By the inner-left convention, the left child has b as its contents, and the right child contains p. (Recall that the left child

does not necessarily correspond to points with smaller coordinates.) Observe that this insertion process generates nodes in pairs, a shrink node whose inner child is a split node. Irrespective of the insertion order, the following invariants are preserved after creation:

- Each cell contains at most one inner box.
- Each leaf cell contains either a single point or a single inner box (unless the tree is empty, in which case it contains nothing).
- The inner child of each shrink node is a split node.

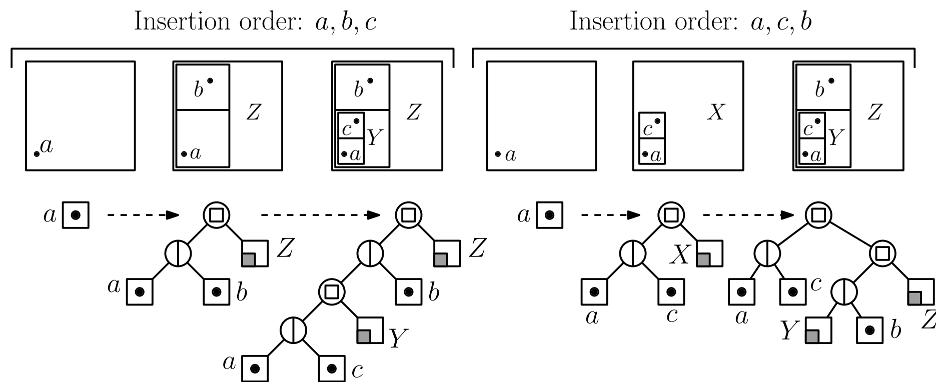


Fig. 3.3: The tree on the left has been created with the sequence $\langle a, b, c \rangle$ and the one on the right with $\langle a, c, b \rangle$. Uppercase letters identify the outer cells of shrink nodes.

An example showing two different insertion orders is given in Fig. 3.3. Although the tree structure depends on the insertion order, the following lemma shows that the subdivision induced by the tree does not. The proof is based on an inductive argument. The key is that, irrespective of the insertion order, the highest level decomposition step (which need not be at the root node) is a function of the smallest quadtree box containing all the points.

Lemma 3.2.1 *Let T be a BD-tree resulting from applying the incremental algorithm to a set P of points. The subdivision induced by the leaf cells of T is independent of the insertion order.*

Proof: The proof proceeds by induction on the size of the tree. Let P denote the set of points inserted into the tree, and let U denote the initial unit hypercube. If $|P| = 1$, the claim holds trivially. Otherwise, fix any permutation of P, and let $\langle p_1, \ldots, p_n \rangle$ denote the resulting insertion sequence. Let E denote the largest shrinking box resulting from this insertion order. (It may be that $E = U$.) Let E_1 and E_2 denote the quadtree boxes that result by splitting E, and let $P_1 = P \cap E_1$ and $P_2 = P \cap E_2$. We will show that E is independent of the insertion order, and hence so are E_1 and E_2 and P_1 and P_2.

Observe first that P_1 and P_2 are disjoint, and no point of P can lie outside of E, since otherwise the insertion of this point would have generated a larger shrinking box. Therefore, P_1 and P_2 form a partition of P. Assuming that each point of P is labeled with its position in the insertion order, that is, its *insertion time*. Let t_1 and t_2 denote the minimum insertion times of P_1 and P_2, respectively. Without loss of generality, we may assume that $t_1 < t_2$. Just prior to the insertion of point p_{t_2}, all the inserted points lie within E_1, and thus, $U \setminus E_1$ is free of any elements of the current subdivision. Since $t_1 < t_2$, there exists at least one point in E_1 at this time, and therefore the leaf cell into which p_{t_2} falls contains either a point of P_1 or an inner box containing the previously inserted points of P_1. Let b denote the contents of this leaf.

The insertion of t_2 into this leaf, produces a shrink to the smallest quadtree box containing both b and p_{t_2}, which, by basic properties of quadtree boxes, is E. Furthermore, the subsequent split creates nodes with outer boxes E_1 and E_2. Following this, all subsequent insertions will be made within either E_1 or E_2, and therefore no larger shrinking box will be produced.

We have shown, therefore, that irrespective of the insertion order, the largest shrink box created during tree creation and the subsequent split are independent of insertion order. (Although the time of their creation is dependent on the insertion order.) By induction, this is true for the insertion of P_1 into E_1 and P_2 into E_2. Therefore the subdivision resulting from the creation process is independent of the insertion order. □

3.2.2 Node Labels and the Heap Property

In this section we show that, if points are labeled with their insertion times, then it is possible to assign labels to the internal nodes of the tree based on the labels of its descendant leaves, so that these labels satisfy a heap ordering. This observation is analogous to the heap condition in the treap data structure of Seidel and Aragon [97].

To motivate this labeling, observe that each newly created internal node is added by the incremental algorithm in order to separate two entities, the newly inserted point and the existing contents of the leaf node in which the new point resides. If we were to label every point with its insertion time (and also provide an appropriate label value to each inner box), it follows that the creation time of any

internal node is the *second smallest* label over all the points in the subtree rooted at this node. This suggests a labeling approach based on maintaining the smallest and second smallest labels of the descendant points.

To make this more formal, we define the following *node-labeling rule*. First, we define two symbols, $\downarrow$ and $\uparrow$ denoting, respectively, numeric values smaller and larger than any insertion times. We create a labeled pair for each node of the BD-tree. First, each leaf that contains a point is labeled with the pair $(t, \uparrow)$, where t is the insertion time of the associated point. Each leaf that contains only an inner box is given the label $(\downarrow, \uparrow)$. For each internal node u, let v and w be its two children, and let $(v^{[1]}, v^{[2]})$ and $(w^{[1]}, w^{[2]})$ denote the labels of its children. If u is a split node it has the label

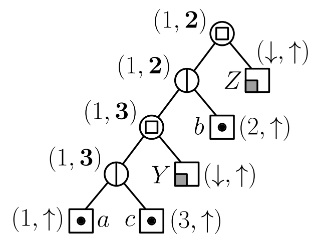


Fig. 3.4: Node labeling for the insertion order $\langle a, b, c\rangle$.

$$\left(u^{[1]}, u^{[2]}\right) \;\leftarrow\; \left(\min\left(v^{[1]}, w^{[1]}\right), \max\left(v^{[1]}, w^{[1]}\right)\right).$$

If u is a shrink node, it is given the label of its inner child (the split node). An example of such a labeling is shown in Fig. 3.4. The following lemma establishes some useful observations regarding the relationships between the labels of nodes in the tree and the insertion times of the points in their associated subtrees.

Lemma 3.2.2 *Consider the labeled BD-tree resulting from the incremental insertion algorithm. For any node u in this tree:*

(*i*) *If u's cell has an inner box, then* $u^{[1]} = \downarrow$, *and in particular, this holds for the*

outer child of any shrink node. Otherwise $u^{[1]}$ *is equal to the earliest insertion time of any point in* u*'s subtree.*

(ii) *If* u *is an internal node, then* $u^{[2]}$ *is equal to the node's creation time, and in particular,* $u^{[2]} \notin \{\downarrow, \uparrow\}$.

(iii) *If* u *is a shrink node, then for every node* v *in its outer subtree* $v^{[2]} > u^{[2]}$.

Proof: We establish (i) by induction. First, suppose that u's cell has an inner box. Clearly, if u is a leaf and it has an inner box, then $u^{[1]} = \downarrow$ by definition. If u is a split node whose cell has an inner box, then this inner box lies within one of its children, call it v. By induction, we have $v^{[1]} = \downarrow$. By the inner-left convention, it follows that v is u's left child. Since $u^{[1]} \leq v^{[1]}$, we have $u^{[1]} = \downarrow$. If u is a shrink node, its label is defined to be the label of its inner child, which is a split node. By our construction the shrinking box contains the inner box, and it follows by induction that the first label of the inner child is $\downarrow$.

Next, suppose that u's cell does not have an inner box. Let t denote the earliest insertion time of any point in the subtree rooted at u. If u is a leaf, then by definition, $u^{[1]} = t$. If u is a split node, then $u^{[1]}$ is the minimum of the first label components of its two children, and so by induction $u^{[1]} = t$. If u is a shrink node, observe first that a point is inserted into its inner subtree (at the moment the shrink node is created) before any point is inserted into its outer subtree. Thus, t is the insertion time of some point in its inner subtree. By definition, $u^{[1]}$ is equal to the first component of its inner child, which is a split node, and, by induction, this is equal to t. This establishes (i).

To establish (ii), consider the insertion of point p at time t. Consider the leaf node whose cell contains p, and let b denote its contents. If b is a point, let t' be the time of b's insertion. Since b already exists, we have $t' < t$. If, on the other hand, b is an inner box, then let $t' = \downarrow$. Since $\downarrow$ is smaller than any insertion time, we have $t' < t$. In either case, if u is the newly created split node, then $u^{[2]} = \max(t', t) = t$, as desired. By parts (i), future insertions cannot increase the first label components of u's two children, and so $u^{[2]} = t$ holds for the final tree. If u is a shrink node, we appeal to the fact that u was created at the same time as its inner child (a split node) and the fact that u's outer child label does not affect u's label. Thus claim (ii) holds.

We know from (ii) that the creation time of any internal node v is $v^{[2]}$. As observed earlier, if v is a shrink node, then every point inserted into its outer child's subtree has a higher insertion time than $v^{[2]}$. For any node w of this subtree, $w^{[2]}$ is the insertion time of such a point, and therefore $w^{[2]} > v^{[2]}$. This establishes (iii).

□

With the aid of this observation, the node labels satisfy the following properties. The second property is called the *heap property*.

Lemma 3.2.3 *The labeled BD-tree resulting from the incremental insertion algorithm satisfies the following properties:*

(i) If v and w are left and right children of a split node, respectively, then $v^{[1]} < w^{[1]}$.

(ii) Given any pair consisting of a parent u and child v, we have $u^{[2]} \leq v^{[2]}$. If u is a split node, the inequality is proper.

Proof: Whenever an insertion is performed, the newly inserted point is placed into a leaf, which is the right child of a newly created split node. The prior contents of the leaf that was split has an earlier insertion time (or $\downarrow$ if the contents were an inner box). Therefore, claim (i) holds for every split node at the time of its insertion. Subsequent insertions cannot alter the first label component of a node, since all such insertions involve points with later insertion times. This establishes claim (i).

To establish claim (ii), we consider two cases. If v is a leaf node, then $v^{[2]} = \uparrow$, and the claim holds trivially. Otherwise, both u and v are internal nodes, and by Lemma 3.2.2(ii), their second label components are their creation times. Clearly, v could not be created before u, since our insertion algorithm adds new nodes only to the bottom of the tree.

If u is a split node, then we assert that $u^{[2]} < v^{[2]}$. Observe first that this holds at the time that u was created, since its two children are leaves, and hence their second component values are equal to $\uparrow$. But by Lemma 3.2.2(ii), we know that $u^{[2]} \neq \uparrow$, and therefore $u^{[2]} < v^{[2]}$. Future insertions may replace v with an internal node, but if so Lemma 3.2.2(ii) implies that $u^{[2]}$ and $v^{[2]}$ are both equal to their creation times. Since such insertions occur strictly after u's creation, we have $u^{[2]} < v^{[2]}$, as desired. □

3.2.3 Randomized Incremental Construction

In this section we show that, if points are inserted in random order into the BD-tree by the above incremental algorithm, with high probability, the resulting tree has $O(\log n)$ height. We begin by showing that the search depth of any query point is $O(\log n)$ in expectation (over all possible insertion orders). The proof is a standard application of backwards analysis [39]. It is based on the observations that (1) the subdivision induced by the tree is independent of the insertion order (as shown in Lemma 3.2.1), and (2) the existence of each cell of the subdivision depends only on a constant number of points. Thus, the probability that any one insertion affects the leaf cell containing the query point is inversely proportional to the number of points inserted thus far.

Lemma 3.2.4 *Consider the BD-tree resulting from incrementally inserting an n-element point set P in random order. Given an arbitrary point q, the expected depth in the tree (averaged over all insertion orders) of the leaf containing q is $O(\log n)$.*

Proof: We will track the cell of the induced subdivision containing q throughout the course of the insertion process. Let Δ_i denote the cell of the induced subdivision containing q after the insertion of the first i points. Observe that if $\Delta_i = \Delta_{i-1}$, then insertion of the ith point did not affect the cell of the subdivision containing q, and therefore q's depth is unaffected by this insertion. If, on the other hand, if $\Delta_i \neq \Delta_{i-1}$, then the insertion of the ith point occurred within q's leaf, and so q may have fallen as many as two levels in the tree. For $1 \leq i \leq n$, let P_i denote the probability, taken over random insertion orders, that $\Delta_i \neq \Delta_{i-1}$. Letting $D(q)$

denote the expected depth of q in the final search tree, we have $D(q) \leq 2\sum_{i=1}^{n} P_i$. We will show below that $P_i \leq \frac{2}{i}$. From this it will follow that the expected depth of q is at most $2\sum_{i=1}^{n} \frac{2}{i} = O(\log n)$.

To show that $P_i \leq \frac{2}{i}$, we apply a backwards analysis. We say that a subdivision cell Δ *depends* on a point p, if p would have caused Δ to be created, had p been inserted last. Consider the subdivision cell c that contains q after the ith insertion.

If c resulted from a split operation, let P_1 and P_2 be the two subsets of points on either side of the split. If either set contains a single point, then c is dependent on this point, since had it not been inserted, there would be no reason for the split. If either set has two or more points, the last insertion of such a point could not have resulted in c's creation, since the other points of the cell would have already resulted in the cell's existence. Thus, c is dependent on at most two points.

If, on the other hand, c resulted from a shrink, let P_1 and P_2 be the two subsets of the split node of the inner child of the shrink node. Again, c could be dependent on at most two points, one from each set. Since, by Lemma 3.2.1, the subdivision does not depend on the insertion sequence so far (just the set of inserted points), and each point is equally likely to have been the last point to be inserted, the probability that the last insertion caused q to change cells is at most $\frac{2}{i}$, which completes the proof. $\square$

The following theorem strengthens the above result by showing that the height bound holds with high probability. A bound on the construction time and space follow immediately. Again, this is a straightforward generalization of analyses of

other randomized incremental algorithms (see, e.g., [39]).

Theorem 3.2.1 *Consider the BD-tree resulting from incrementally inserting an n-element point set in random order. The tree has space $O(n)$ (unconditionally). With high probability, the height of the tree is $O(\log n)$, and the time needed to construct the tree is $O(n \log n)$.*

Proof: (of Theorem 3.2.1) The $O(n)$ space bound follows trivially from the fact that each insertion creates $O(1)$ new nodes. To bound the height of the tree, we will show that, for any positive parameter λ, the probability that the height of the tree exceeds $\lambda \ln(n+1)$ is at most $1/n^{\Omega(\lambda)}$. We define a directed acyclic graph G, with one source and one sink. Paths in G from the source to the sink correspond to the permutations of P. There is a node for every subset of P, including one for the empty set. It is convenient to imagine the nodes as being grouped into $n+1$ layers, such that layer i contains the subsets of cardinality i. The node corresponding to a subset P' has an outgoing arc to the node corresponding to a subset P'' if P'' can be obtained by adding one point of P to P'. We label the arc with this point. Note that the node corresponding to P' in layer i has exactly i incoming arcs, each labeled with a point in P, and exactly $n-i$ outgoing arcs, each labeled with a point $S \setminus S'$.

Directed paths from the source to the sink in G now correspond one-to-one to permutations of P, which correspond to possible executions of the construction. Consider an arc of G from the node corresponding to P' in layer i to the node corresponding to P'' in layer $i+1$. Let the point p be the label of the arc. We

mark this arc if the insertion of p changes the cell containing point q. To be able to say something about the number of marked arcs, we use the same backwards analysis argument from the proof of Lemma 3.2.4. There are at most two points whose insertion would change the cell containing q when they are removed from a subset P''. This means that any node in G has at most two marked incoming arcs.

We want to analyze the expected number of marked edges on a source-to-sink path in G. To do this we define the random variable X_i to be 1 if the i-th arc on the sink-to-source path in G is marked, and 0 otherwise. Since each node in layer i has i incoming arcs, at most two of which are marked, this implies that $Pr[X_i = 1] \leq \frac{2}{i}$.

Let $Y = \sum_{i=1}^{n} X_i$. Since a query point can fall at most two levels with each change of its containing cell, the number of nodes on the search path for q is at most $2Y$. We will bound the probability that Y exceeds $\lambda \ln(n+1)$. By Markov's inequality, for any $t > 0$ we have

$$Pr[Y \geq \lambda \ln(n+1)] \;=\; Pr\left[e^{tY} \geq e^{t\lambda \ln(n+1)}\right] \;\leq\; e^{-t\lambda \ln(n+1)} E\left[e^{tY}\right].$$

Since the random variable X_i is independent, we have

$$E[e^{tY}] \;=\; E\left[e^{\sum_{i=1}^{n} tX_i}\right] \;=\; E\left[\prod_{i=1}^{n} e^{tX_i}\right] \;=\; \prod_{i=1}^{n} E\left[e^{tX_i}\right].$$

By setting $t = \ln 1.5$, we obtain

$$\begin{aligned}\prod_{i=1}^{n} E[e^{tX_i}] &\leq \prod_{i=1}^{n}\left(e^t\frac{2}{i} + e^0\left(1-\frac{2}{i}\right)\right) = \prod_{i=1}^{n}\left(1+\frac{1}{i}\right) = \prod_{i=1}^{n}\frac{1+i}{i} \\ &= \frac{2}{1}\frac{3}{2}\cdots\frac{n+1}{n} = n+1.\end{aligned}$$

Putting everything together, we obtain

$$Pr[Y \geq \lambda \ln(n+1)] \leq e^{-\lambda t \ln(n+1)}(n+1) = (n+1)^{-\lambda t}(n+1) = \left(\frac{1}{n+1}\right)^{\lambda t - 1},$$

which is $1/n^{\Omega(\lambda)}$, as desired.

Since the depth of an arbitrary point is $O(\log n)$ with high probability, the total search time for all the points of P is $O(n \log n)$, also with high probability, and this is the dominant term in the tree's construction time. □

3.3 The Quadtreap

In this section we define our dynamic data structure, which we call the *quadtreap*. Before presenting the algorithm, we discuss the basic rebalancing operation upon which the data structure is based.

3.3.1 Pseudo-nodes and Rotation

The BD-tree depends on the insertion order, but we will show in this section that it can be rebalanced through a simple local operation, which we refer to variously as

rotation or *promotion*. This operation is much like the familiar rotation operation defined on binary search trees, but some modifications will be needed in the context of BD-trees. This operation will be a central primitive in maintaining balance in our data structure as points are inserted and deleted.

Throughout this section we assume that the BD-tree satisfies the property that the internal nodes of the BD-tree can be partitioned into pairs, such that each pair consists of a shrink node as a parent and a split node as its inner child. We call this the *shrink-split property*. As mentioned in Section 3.2, the BD-tree resulting from our incremental construction algorithm satisfies this property.

This assumption allows us to view each shrink-split combination as single decomposition operation, which subdivides a cell into three subcells. Thus, we can conceptually merge each shrink-split pair into a single node, called a *pseudo-node* (see Fig. 3.5), which has three children. The first two children, called the *left* and *right child*, correspond to the children of the split node, and the third child, called the *outer child*, corresponds to the outer child of the shrink node. Note that this is a conceptual device and does not involve any structural changes to the tree.

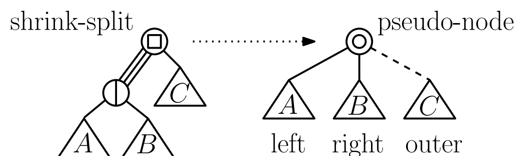


Fig. 3.5: The pseudo-node corresponding to a shrink-split pair.

Let us now define rotation on pseudo-nodes. We call the operation a *promotion* and it is specified by giving a node y of the tree that is either a left child or an outer child. Let x denote y's parent. If y is a left child, the operation promote(y) makes x

the outer child of y, and y's old outer child becomes the new left child of x. Observe that the inner-left convention is preserved, since y's left child is unchanged, and x's left child (labeled w in the figure) contains the inner box (consisting of the union of A and B's cells in the figure).

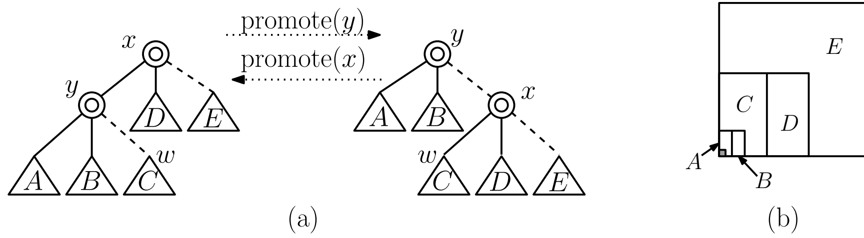


Fig. 3.6: Rotation operation on pseudo-nodes (a). The associated subdivision is shown in (b). (The order of left-right children is based on the assumption that the earliest insertion time (or inner box, if present) lies within the cell associated with subtree A.)

Next, let us consider the promotion of an outer child x. Let y denote its parent. The operation promote(x) is the inverse of the previously described left-child promotion. It makes y the left child of x, and x's old left child becomes the new outer child of y. Again, the inner-left convention is preserved, since prior to the promotion, if y's cell had an inner box it must lie in its left child A, and so after promotion it lies in x's left child y. (The reason for the inner-left convention is to allow us to define one promotion rule. Otherwise, there would need to be a third promotion rule involving the right-child edge.)

Based on the above descriptions, it is easy to see that the essential properties of the BD-tree are preserved after applying either promotion operation. Note that promotion does not alter the underlying spatial decomposition, just the tree structure. Since it is a purely local operation, promotion can be performed in $O(1)$

time.

3.3.2 Point Insertion

We have seen that, if the points are inserted in random order, the height of the BD-tree is $O(\log n)$ with high probability, but a particularly bad insertion order could result in a tree of height $\Omega(n)$. In order to make the tree's height independent of the insertion order, we assign each point a random numeric *priority* at the time it is inserted, and we maintain the tree *as if* the points had been inserted in increasing priority order. This is essentially the same insight underlying the treap data structure [97]. Intuitively, since the priorities are not revealed to the user of the data structure, the tree effectively behaves as if the insertion order had been random.

The tree's structure is determined by the node-labeling rule described in Section 3.2.2 and the properties of Lemmas 3.2.2 and 3.2.3. Recall that the labeling procedure of that section gives both nodes of each shrink-split pair the same label. We define the label of each pseudo-node to be this value. Whenever a new point is inserted into or deleted from the tree, we need to restructure the tree so that its structure is consistent with the heap property. Thus, the tree behaves *as if* the nodes of the tree had been inserted in priority order. We begin with the insertion algorithm.

A newly inserted point is assigned a random priority $0 < t < 1$. It is then inserted into the tree, by the incremental algorithm. The newly created nodes (three leaves, one split node, and one shrink node) generated by the incremental algorithm

are labeled as if the insertion took place at time t. The resulting tree may violate the heap properties, however, and the tree needs to be restructured to restore this property. The restructuring algorithm is presented in Algorithm 1. Recall that the label of a node u is denoted by $(u^{[1]}, u^{[2]})$. The restructuring procedure makes use of the following simple utility functions:

- adjust-left-right(u): Let v and w be the left and right children of u, respectively. If $w^{[1]} < v^{[1]}$, swap these children. (This enforces property (i) of Lemma 3.2.3.)
- update-label(u): Let w_1 and w_2 denote u's left and right children, respectively, Set

$$(u^{[1]}, u^{[2]}) \leftarrow ({w_1}^{[1]}, {w_2}^{[1]}).$$

Finally, the operation update(u) invokes both of these functions on u.

Algorithm 1 Tree restructuring after inserting a point in a new leaf node v.

```
1:  while v ≠ root(T) do
2:      u ← parent(v)
3:      if v is a left or right child then
4:          update(u)
5:          if v^[2] < u^[2] then
6:              promote(v),  update(u)
7:          else v ← u
8:      else if v^[2] < u^[2] then
9:          promote(v),  update(v)
10:     else break
```

An example of the insertion and subsequent restructuring is shown in Fig. 3.7. (For ease of illustration, we assume that priorities are positive integers.) Consider the insertion of a point with priority 2, which lies within the leaf node indicated in

Fig. 3.7(a). This insertion creates a pseudo-node v with label $(\downarrow, 2)$, which violates Lemma 3.2.3(ii) with respect to its parent u (see (b)), thus causing a promotion. After the promotion, there is a violation of Lemma 3.2.3(ii) between v and its new parent, whose label is $(\downarrow, 6)$ (see (c)), resulting in another promotion. There is now violation of the heap property with the root node (see (d)), causing a final promotion. At this point v is the root, and we obtain the final tree (see (e)).

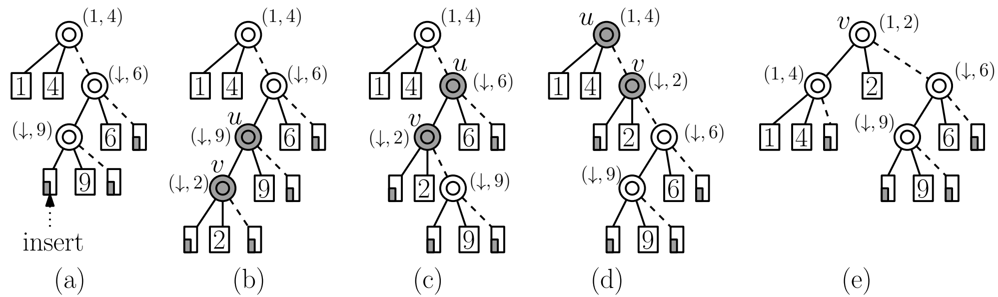


Fig. 3.7: Example of quadtreap restructuring after insertion.

The following lemmas establish the correctness and running time of this procedure. The proof involves a straightforward induction involving a case analysis of the relationships between the labels of nodes before and after each promotion. Together, they establish Theorem 3.1.1(i).

Lemma 3.3.1 *After inserting a point and applying Algorithm 1, the quadtreap satisfies Lemma 3.2.3.*

Proof: Let us assume that Lemma 3.2.3 was satisfied prior to the insertion. Let t denote the priority of the newly inserted point. Consider an arbitrary iteration of the while loop of Algorithm 1. We assert that the subtree rooted at v satisfies the properties of Lemma 3.2.3. To see that this suffices, observe that the algorithm

terminates either (1) when v is the root or (2) when v is an outer child and $v^{[2]} \geq u^{[2]}$. In the former case, our assertion applies to the entire tree. In the latter case, since v is an outer child, u's label is not affected by v's label (and hence neither are any of u's ancestors). Therefore, Lemma 3.2.3 holds for the subtree rooted at v (by the assertion), and they hold throughout the rest of the tree (since (u, v) is satisfied, and no other labels have been changed).

To establish the assertion, consider an arbitrary iteration of the while loop. Let v be the current node, and let u be its parent. We consider two cases: (1) v is a left or right child, and (2) v is an outer child.

Case 1 (v is a left or right child): Let w be other left-right child of u, and let x be v's outer child (see Fig. 3.8(a)). The call to update(u) on line 4 guarantees that claim (i) of Lemma 3.2.3 holds for v and w. (Note that there is no change to u's label, since it is a symmetric function of its left and right children.) If $v^{[2]} \geq u^{[2]}$, then v does not violate claim (ii) of Lemma 3.2.3 with respect to u. In this case, the algorithm sets v to u and continues with the next iteration.

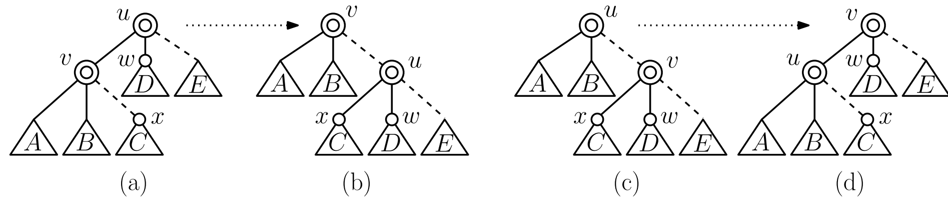


Fig. 3.8: Proof of Lemma 3.3.1.

On the other hand, if $v^{[2]} < u^{[2]}$, then Lemma 3.2.3(ii) is violated. First, we

claim that $v^{[1]} < w^{[1]}$. To see this, observe that otherwise we would have

$$v^{[1]} \leq v^{[2]} < u^{[2]} = \max(v^{[1]}, w^{[1]}) = v^{[1]},$$

which is clearly impossible. By this claim, the call to adjust-left-right resulting from update(u) on line 4 guarantees that v is the left child, and w is the right child. Since $u^{[2]} = \max(v^{[1]}, w^{[1]})$, we have $u^{[2]} = w^{[1]}$. After applying the right rotation on line 6, v is now the parent of u, and u now has x as its left child (see Fig. 3.8(b)).

After the call to update(u) on line 6, claims (i) and (ii) of Lemma 3.2.3 are established among u and its children, but u's label may have been modified. We claim that the newly assigned second label component, call it $u'^{[2]}$, cannot violate claim (ii) of Lemma 3.2.3 with respect to v. To see this, observe that

$$u'^{[2]} = \max(x^{[1]}, w^{[1]}) = \max(x^{[1]}, u^{[2]}) \geq u^{[2]} > v^{[2]}.$$

Because v's left and right children have not changed, they still satisfy Lemma 3.2.3(i). Thus, the new subtree rooted at v satisfies Lemma 3.2.3(i) and (ii), as desired.

Case 2 (v is an outer child): If $v^{[2]} \geq u^{[2]}$, there is no violation of Lemma 3.2.3(ii). Because v's label does not affect u's label (and hence it does not affect any of the other tree labels), the algorithm may terminate at this point.

If, on the other hand, $v^{[2]} < u^{[2]}$, then let x and w denote the left and right children of v, respectively (see Fig. 3.8(c)). By Lemma 3.2.3(i) we have $x^{[1]} < w^{[1]}$,

and so

$$v^{[2]} = \max(x^{[1]}, w^{[1]}) = w^{[1]}.$$

After the call to promote(v) on line 9, v is now the parent of u, and u has now acquired x as its new outer child (see Fig. 3.8(d)).

Prior to the insertion, v's second label component was not smaller than u's second label component. The fact this is now violated implies that $w^{[1]}$ has decreased as a result of the insertion, and hence its value is the priority of the newly inserted point. In particular, this means that the insertion did not occur in the subtree rooted at x. Prior to the insertion, v was u's outer child, and since x is a node of this outer subtree, by Lemma 3.2.2(iii) we have $x^{[2]} \geq u^{[2]}$. Since x is the only child of u to change, Lemma 3.2.3(ii) is now satisfied at u. Since u's left and right children have not changed, Lemma 3.2.3(i) is still satisfied at u.

After the call to update(v) on line 9, Lemma 3.2.3(i) will be satisfied at v. Because $v^{[2]} < u^{[2]}$, Lemma 3.2.3(ii) is satisfied at v as well. Therefore, the new subtree rooted at v satisfies Lemma 3.2.3(i) and (ii), as desired. □

Lemma 3.3.2 *The time to insert a point in a quadtreap of size n and height h is $O(h)$. Thus, the insertion time is $O(\log n)$ with high probability.*

Proof: The promotion and updating of priority values can be performed in $O(1)$ time per node. The time needed to locate the point to be inserted is $O(h)$. The time to insert the point is $O(1)$. Since the algorithm traces the path from the insertion point to the root, the time to perform the restoration of the heap properties by

Algorithm 1 is $O(h)$. Thus, the entire insertion time is $O(h)$. By Theorem 3.2.1, $h = O(\log n)$ time with high probability. □

3.3.3 Point Deletion

Next, we consider the deletion of a point, p. We first determine the leaf node v containing this point. We handle deletion by reversing the insertion process. In particular, we effectively set the deleted point's priority to $\uparrow$ by assigning v a label of $(\uparrow, \uparrow)$. We then restructure the tree, described below, so that the properties of Lemmas 3.2.3 are satisfied. Because of p's high priority, the tree's structure would be the same as if p had been the last point to be inserted. Viewing from the perspective of pseudo-nodes, when a point is inserted into the tree, it is placed within a leaf cell that is the right child of its parent, and its two siblings are both leaves. To establish this claim, observe first that v cannot be an outer child because the cell associated with an outer child has an inner box. Since v contains a point, it cannot also contain an inner box. Also, v cannot be a left child since this would violate Lemma 3.2.3(i). Therefore, v is the right child of its parent. To show that v's siblings are leaves, note that by the node-labeling rule, v's parent's second label component is $\uparrow$. If its siblings were not leaves, their second label components would be strictly smaller than this, thus violating property (ii) of Lemma 3.2.3. Therefore, after restructuring, we can delete p by simply "undoing" the insertion procedure by replacing the subtree rooted at v's parent with a single leaf node. All that remains is to describe the above restructuring process. This is given in Algorithm 2.

Algorithm 2 Restructuring the tree as part of the deletion of a point in a leaf v.

1: label(v) $\leftarrow (\uparrow, \uparrow)$
2: $r \leftarrow$ parent(v)
3: **while** $r \neq$ null **do**
4: $\quad u \leftarrow r,\ \ r \leftarrow$ parent(r)
5: $\quad$ update(u)
6: $\quad w \leftarrow$ left(u), $\ x \leftarrow$ outer(u)
7: $\quad$ **while** $w^{[2]} < u^{[2]}$ or $x^{[2]} < u^{[2]}$ **do**
8: $\quad\quad$ **if** $w^{[2]} < x^{[2]}$ **then**
9: $\quad\quad\quad$ promote(w), update(u)
10: $\quad\quad\quad w \leftarrow$ left(u)
11: $\quad\quad$ **else**
12: $\quad\quad\quad$ promote(x), update(x)
13: $\quad\quad\quad x \leftarrow$ outer(u)

An example of the restructuring for the deletion process is shown in Fig. 3.9. (For ease of illustration, let us assume that priorities are positive integers.) The point to be deleted lies in node v (see (a)). We set its label to $(\uparrow, \uparrow)$, set u to v's parent and then invoke the update procedure on line 5 of Algorithm 2. As a result, u's left and right children are swapped, and u's label is updated (see (b)). A sequence of promotions (actually "demotions") is then applied to restore the heap properties (see (c)–(e)). Finally, the subtree rooted at v's parent is replaced by a single leaf node (see (f)).

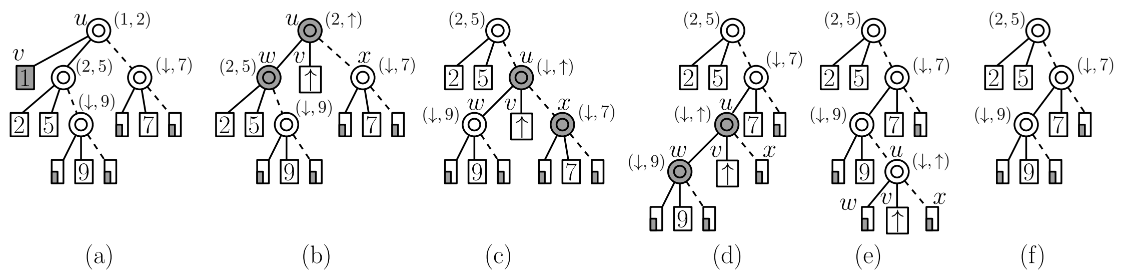


Fig. 3.9: Example of deletion from a quadtreap.

The following lemmas establish the correctness and running time of this pro-

cedure.

Theorem 3.3.1 *After deleting a point and applying the restructuring procedure of Algorithm 2, the quadtreap satisfies Lemma 3.2.3.*

Proof: The algorithm begins by setting the priority of the leaf v to $(\uparrow, \uparrow)$ and assigning r to be v's parent. The algorithm then enters the outer while loop, which assigns u to r, and r to its parent, and then proceeds to restructure the subtree rooted at u, which we call *local restructuring*.

This local restructuring will generally alter this subtree's root, and hence may violate the conditions of Lemma 3.2.3 with respect to r. Thus, the outer loop repeatedly invokes the local restructuring until arriving at the root. (In fact, the algorithm could terminate as soon as the subtree's root label is unchanged following the local restructuring, but this would not alter the worst-case running time.) To complete the proof, it suffices to establish the correctness of the local restructuring.

The effect of setting v's label to $(\uparrow, \uparrow)$, results in the second label component of its parent u to be set to $\uparrow$. In general, in subsequent iterations of the outer loop, the first label of the restructured subtree can only increase. The call to update(u) on line 5, results in an increase of u's label components, and this may result in a violation of Lemma 3.2.3(ii), with respect to the siblings of the node with the increased label. If there is such a violation after the call to update, the violated siblings will be the left child w or the outer child x of u (since the node with the larger first label is made the right child after invoking update(u)).

Consider the sequence of nodes consisting of the iterated outer children of w

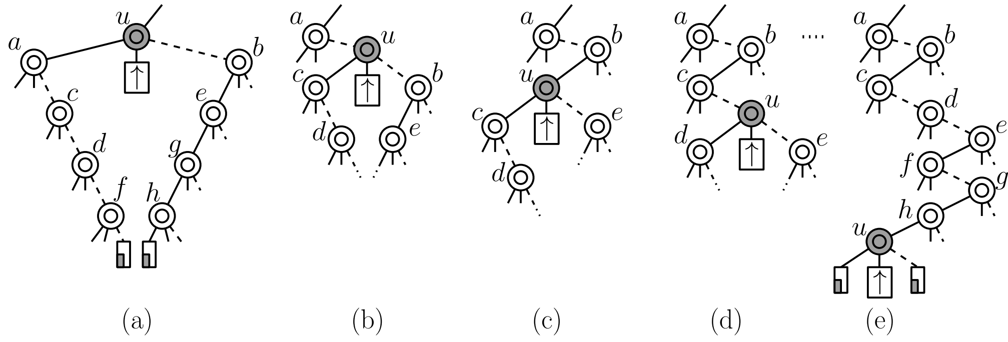


Fig. 3.10: Illustrating the execution of the inner loop of Algorithm 2.

and the sequence of the iterated left children of x (see Fig. 3.10 (a)). The effect of the inner while loop is to repeatedly rotate these nodes up one at a time, while rotating u down, until u's left and outer children both satisfy Lemma 3.2.3(ii) with respect to u. The order in which the nodes are rotated depends on the values of their second label components. Each time through the inner while loop, the node at the head of each sequence (either w or x) having the smaller second-label value is rotated into u's position. (The effect of these rotations is illustrated in Fig. 3.10. Part (a) shows the initial configuration, where we assume that the alphabetical ordering of the node names $\{a, b, \ldots, h\}$ correspond with an increasing enumeration of second-label components.)

It follows that, after the inner loop terminates, the nodes along the path from u up to r satisfy Lemma 3.2.3(ii). Except along the path from u, the nodes involved in the rotations keep their original children, and therefore they satisfy Lemma 3.2.3(ii) as well. After each right rotation, node u obtains a new left child, and so the subsequent call to update(u) enforces Lemma 3.2.3(i). Similarly, after each left rotation, node x obtains a new left child, and the subsequent call to update(x)

enforces Lemma 3.2.3(i). Therefore, on termination, the conditions of Lemma 3.2.3 are satisfied throughout the subtree, as desired. □

Deletion requires $O(\log^2 n)$ time. Intuitively, the reason that deletion is more complicated than insertion is because the deleted node's priority may exist among the label components along the entire search path to the root. (Consider, for example, the deletion of the point with the lowest priority in the tree.) Deleting the point involves updating the label of each such node. Each update may result in $O(\log n)$ promotions.

Lemma 3.3.3 *The time to delete a point from a quadtreap of size n and height h is $O(h^2)$. Thus, the deletion time is $O(\log^2 n)$ with high probability.*

Proof: The promotion and updating of each label can be performed in time $O(1)$. Each iteration of the inner loop of Algorithm 2 takes time proportional to the height of node u. This inner loop may be repeated for each ancestor of the leaf v containing the deleted point. Therefore, the total deletion time is at most $O(h^2)$. Since the depth of the tree by Theorem 3.2.1 is $O(\log n)$ with high probability, the deletion time is $O(\log^2 n)$, also with high probability. □

The worst case scenario for point deletion is rather pessimistic. The following lemma shows that the expected time to delete a random point of the tree is linear in the tree's height. The expectation is over the choices of which point to delete, whereas the high-probability bound is over the assignment of priorities. This establishes Theorem 3.1.1(ii).

Lemma 3.3.4 *The expected time to delete a random point from a quadtreap T of size n and height h is $O(h)$. Thus, the expected deletion time is $O(\log n)$ with high probability*

Proof: Before giving the proof, we begin with a useful observation. Consider a node u visited in an arbitrary iteration of the outer while loop of Algorithm 2 during the deletion of some point p. We claim that the inner loop will be entered only if p's priority is among the two smallest in the subtree rooted at u. First, observe u is an ancestor of the leaf containing p. The second component of u's label is equal to the second smallest priority in u's subtree, which implies that this component's value can be changed only if p has one of the two smallest priorities in u's subtree. If this observation holds for p and u, p applies a charge to u.

Define an indicator variable, $\chi(u, p)$ as follows.

$$\chi(u,p) \;=\; \begin{cases} 1 & \text{if } p \text{ charges } u, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the expected deletion time t is

$$E(t) \;\leq\; \frac{1}{n} \sum_{p \in P} \sum_{u \in T} height(u) \cdot \chi(u,p).$$

Each node is charged by at most two points, and thus,

$$E(t) \;\leq\; \frac{1}{n} \sum_{u \in T} 2 \cdot height(u) \;\leq\; \frac{1}{n} \cdot n \cdot 2h \;=\; O(h) \;=\; O(\log n).$$

□

3.4 Approximate Range Queries

In this section we present an algorithm for answering approximate range-counting queries using the quadtreap. Recall the problem description from Chapter 1. Because promotions will not be needed here, most of the description here will be based on the simpler representation of the tree as a BD-tree with split and shrink nodes.

In order to provide efficient response to queries, we will need to add a couple of enhancements to the basic data structure. Before discussing these enhancements, let us recall that we are given a set P of n points $\mathbb{R}^d$, for fixed d. Let $\mathrm{wgt}(p)$ denote the weight of point p. Recall that a query is presented in the form of two convex shapes, an inner range and outer range, Q^- and Q^+, respectively, whose boundaries are separated by a distance of at least $\varepsilon \cdot \mathrm{diam}(Q)$. Let T be a BD-tree storing these points, and let h denote its height. For the sake of generality, we will express running times (deterministically) in terms of n, h, and ε, but as shown in the previous sections, if the BD-tree is a quadtreap, then h will be $O(\log n)$ with high probability. Our best results are for range counting where the weights are drawn from a commutative group, but we will also consider the case of commutative semigroups and range reporting.

The first enhancement to the tree is to maintain at each node v the total weight of the points in the associated subtree. Given a node v, let P_v denote the points of P lying within v's subtree, and let $\mathrm{wgt}(v)$ denote their total weight. Observe

that this sum can be updated as points are inserted or deleted, without altering the construction time or space bounds.

The second enhancement is a structural addition to the tree. To motivate the addition, it is useful to recall how approximate range queries are answered in partition trees. We apply a standard top-down recursive algorithm (see for example [12]). Starting at the root, let v denote the current node being visited. We compare v's cell C to the range. If the cell lies outside the inner range or inside the outer range or is a leaf, we can process it in $O(1)$ time. Otherwise, we recurse on its children. (See the code block.)

Algorithm 3 Range Query

Input: A range Q, given with an inner range Q^- and an outer range Q^+, and a node v in the Quadtreap

Output: The total weight of points in the approximate range

```
1: function rangeQuery(Q, v)
2:     C ← cell(v)
3:     if C ∩ Q⁻ = ∅ then return 0
4:     else if C ⊆ Q⁺ then return wgt(v)
5:     else if v is a leaf then
6:         if v contains a point in Q then return wgt(p)
7:         else return 0
8:     else if v is a split node then
9:         return rangeQuery(Q, left(v)) + rangeQuery(Q, right(v))
10:    else                                              ▷ v is a shrink node
11:        return rangeQuery(Q, inner(v)) + rangeQuery(Q, outer(v))
```

The key issue in analyzing the algorithm is determining the number of internal nodes that are *expanded*, which means that a recursive call is made to the node's two children. Recall that the *size* of a cell in the tree is the maximum side length of its outer box. The analysis classifies expanded nodes as being of two types: *large nodes* are those whose cell size is at least $2 \cdot \mathrm{diam}(Q)$, and remainder are called

small nodes. To understand the issue, recall the analysis of the small expanded nodes given in [12]. It relies on an crucial property of the BBD-tree, namely that whenever the search descends a constant number of levels, the size of the associated cells decreases by a constant factor. This property holds for the binary quadtree, since with each d levels of descent in the tree, every side is bisected and so the cell size is halved. However, this generally does not hold for BD-tree. The problem arises from the fact that a the outer child of a shrink node has the same outer box as its parent. Thus, there may arbitrarily long chains of shrink nodes, for which the cell size does not decrease. We define a *shrink chain* to be a maximal sequence of shrink nodes, where each node is the outer child of its predecessor.

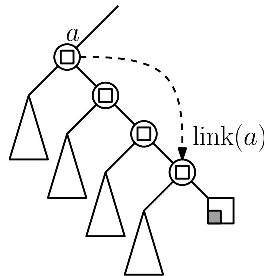


Fig. 3.11: A link.

We fix this problem by adding a special pointer, called a *link*, which points from the first node of each shrink chain to its last node (see Fig. 3.11). We also assume that each node of the tree stores a pointer to its parent. A BD-tree with all the above enhancements (internal weights, parent pointers, and links) is called an *augmented BD-tree*. A quadtreap based on such a tree is called an *augmented quadtreap*.

In Section 3.3.1 we discussed promotions. It is easy to modify the promotion process to maintain node weights and links. To see this, let us briefly return to the pseudo-node view of the tree. Consider the operation promote(y), where y is the left child of some node x (recall Fig. 3.6). Let w be y's outer child. The associated weights wgt(x) and wgt(y) can be updated by assigning each the sum of the weights of their new children. The links are updated as follows. Prior to the promotion, x and y may each have been the head of some shrink chain. Therefore, each may have a link. If y was the first node of a shrink chain prior to the promotion, w is now the head of this chain, and so w is assigned y's old link. If x was the first node of a shrink chain prior to the promotion, y now becomes the first node of the chain after the promotion, and so y is assigned x's old link. The modifications for the promotion of an outer child are symmetrical. (In particular, x is assigned y's old link, and y is assigned w's old link.)

Let us see how the link is used in answering range searching. Given an augmented tree, when the search arrives at the start of a shrink chain and the node is small, rather than descending the chain from top to bottom, as the above search algorithm would, it will traverse the chain from bottom to top. Why does this work? Recall from Section 3.2 our invariant that, if the cell of a shrink node has an inner box, the shrinking box properly contains this inner box. This implies that, when considered from bottom to top, the sizes of the inner boxes decrease monotonically. This is exactly the property we desire when visiting small nodes.

There is a simple and elegant way in which to implement the modified search algorithm. The algorithm is exactly as described above, but whenever we visit a

small shrink node, we “redefine” the meanings of the various tree primitives for each node: left, right, cell, and weight. To motivate this modification, observe that it is possible to transform each shrink chain, into one in which each node in the transformed sequence is the inner child of its parent (see Fig. 3.12). Let T^* denote the resulting transformed tree.[3] Note that this transformed tree exists merely for conceptual purposes (we do not actually modify the tree).

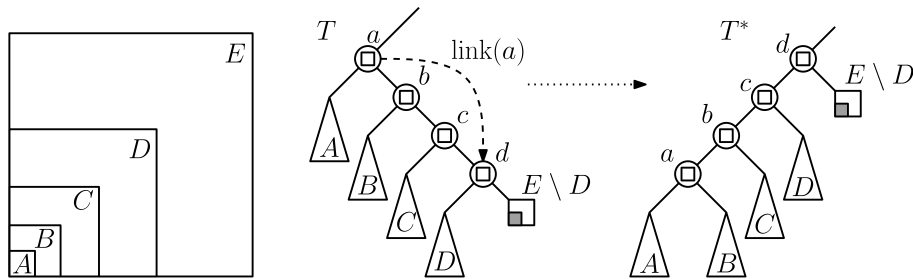


Fig. 3.12: A shrink chain and the transformed tree.

As mentioned above, this transformation can be achieved “on the fly” by appropriately redefining the primitive tree operations as the range search algorithm is running. The algorithm proceeds as before, but whenever a shrink node is first encountered, we save this node, which is called the *head* of the chain (labeled “a” in Fig. 3.12). We then immediately follow its link to obtain the next node to be visited by the search. As long as we continue to visit shrink nodes, we apply the alternate definitions of the various tree primitives given in Fig. 3.13. On returning to a leaf or split node, we return to the standard definitions. Note that the modified weight

[3]The conversion of each shrink chain can be viewed as arising from a series of promotions to the nodes of the shrink chain, but, since we are not considering nodes at the level of pseudo-nodes here, the resulting transformed tree does not satisfy the shrink-split property. Nonetheless, it is equivalent to the original tree from the perspective of the subdivision of space that it defines, and thus it is a valid BD-tree. This is the only property that the range-search procedure requires.

function requires the use of a subtraction operator.

$$\begin{aligned}
\text{inner}^*(v) &\equiv (v = \textit{head}) \;?\; \text{inner}(v) : \text{parent}(v)\\
\text{outer}^*(v) &\equiv (\text{outer}(v) \text{ is a leaf}) \;?\; \text{outer}(v) : \text{inner}(\text{outer}(v))\\
\text{cell}^*(v) &\equiv (\text{outer}(v) \text{ is a leaf}) \;?\; \text{cell}(\textit{head}) : \text{cell}(\textit{head}) \setminus \text{cell}(\text{outer}(\text{outer}(v)))\\
\text{wgt}^*(v) &\equiv (\text{outer}(v) \text{ is a leaf}) \;?\; \text{wgt}(\textit{head}) : \text{wgt}(\textit{head}) - \text{wgt}(\text{outer}(\text{outer}(v)))
\end{aligned}$$

Fig. 3.13: Redefined tree primitives.

For example, in Fig. 3.12, in the tree T^* the outer child of b is C, which is equivalent to inner(outer(b)) in the original tree T, and this matches the definition of outer$^*(b)$.

Because the search algorithm does not rely on the shrink-split property, establishing correctness of the modified range-search algorithm involves showing that the modified primitives achieve the desired transformation.

Lemma 3.4.1 *Given a set of points with weights drawn from a commutative group and an augmented BD-tree storing these points (which need not satisfy the shrink-split property), the modified query algorithm correctly returns a count that includes all the points lying inside the inner range and excludes all the points lying outside the outer range.*

Proof: We exploit the equivalence between the transformed tree and the original tree, and analyze the search algorithm from the perspective of the transformed tree T^*. Let E be the set of nodes visited for which the algorithm does not make a recursive call to its children. It follows by an easy inductive argument that the set of the cells corresponding to nodes of E forms a disjoint union of the hypercube

containing all data points.

We first show that all the points lying inside the range are included in the final count. Let p be some point lying inside the inner range Q^-, and let v be the node of E that contains it. If v is a leaf node, p would be included in the count since it lies within Q. If not, the fact that no recursive call was made implies that $\text{cell}(v) \subseteq Q^+$, and hence p's weight would have been included in the final count.

In an analogous way, we can also show that all the points outside the outer range are not included in the final count. Let p be some point lying outside the outer range Q^+, and let v be the node of E that contains it. As before, the case of a leaf node is trivial. If v is a non-leaf node, the fact that we did not make a recursive call implies that the intersection of $\text{cell}(v) \cap Q^- = \emptyset$ must be empty, and hence p's weight would not have been included in the final count. $\square$

The transformed tree T^* satisfies the property that the cell sizes decrease by a constant factor with every constant number of levels of descent. The original tree T is of height h. It follows from a straightforward generalization of the analysis of [12] (using T for the large expanded nodes and T^* for the small expanded nodes) that the query time is $O(h + (1/\varepsilon)^{d-1})$. Because the use of wgt^* requires subtraction, this query time applies only if the weights are drawn from a group. In the case of semigroup weights, we perform the entire query in T. In the absence of the size-decreasing property, the query time that results is the product (not the sum) of h and $(1/\varepsilon)^{d-1}$. (Answering an approximate range query can be reduced to $O((1/\varepsilon)^{d-1})$ cell queries in T [45], each of which takes $O(h)$ time.)

The above query algorithm (for the group case) can be adapted to answer range reporting queries. Whenever we encounter a node whose cell lies entirely inside Q^+, rather than returning the sum of the weights of the points in this node, the algorithm traverses the associated subtree and enumerates all its points.

Summarizing all of these observations, we have the following result, which establishes Theorem 3.1.1(iii) and (iv).

Theorem 3.4.1 *Consider a set of n points in $\mathbb{R}^d$ that have been stored in an augmented BD-tree of height h. Then, for any $\varepsilon > 0$ and any convex range Q satisfying the unit-cost assumption:*

(i) If the point weights are drawn from a commutative group, it is possible to answer ε-approximate range counting queries in time $O(h + (1/\varepsilon)^{d-1})$.

(ii) If the point weights are drawn from a commutative semigroup, it is possible to answer ε-approximate counting queries in time $O(h \cdot (1/\varepsilon)^{d-1})$.

(iii) It is possible to answer ε-approximate range reporting queries in time $O(h + (1/\varepsilon)^{d-1} + k)$, where k is the number of points reported.

Proof: Let Q denote the query range and let T denote the augmented BD-tree. We start with some definitions. An internal node v is said to be *visited* if a recursive call is made on it. It is said to be *expanded* if it is visited, and a recursive call has been made to its children. Let $w = \text{diam}(Q)$. We distinguish between two kinds of expanded nodes depending on the size (longest side length) of the associated cell. An expanded node v whose size is at least $2w$ is *large*, and otherwise it is *small.*

We will show that the number of large expanded nodes is bounded by $O(h)$, and the number of small expanded nodes is bounded by $O((1/\varepsilon)^{d-1})$. Since each node can be expanded and its children visited in constant time, it follows that the total running time is the sum of these two quantities.

We first bound the number of large expanded nodes. Recall that the transformed primitives apply only to the small expanded nodes, and hence we may assume that we are dealing with the original augmented BD-tree. Consider the set of all expanded nodes of size greater than $2w$. These nodes induce a subtree of the BD-tree. Let L denote the leaves of this subtree. The cells associated with the elements of L have pairwise disjoint interiors, and they all intersect the range (otherwise they would not have been expanded). By the packing lemma of [12], there are $O(1)$ such boxes. Thus, the total number of large expanded nodes is $O(h)$.

Next, we consider the small expanded nodes. In processing these nodes, we consider that we are traversing the transformed tree T^*. We show first that the total number of small expanded nodes is not greater than a constant times the number of small split and leaf nodes that are visited. To see this, consider a small shrink node v that is expanded by the search. Such a node will be traversed in the transformed tree T^*. By definition of outer$^*(v)$, if v's outer child in T is a leaf, then this leaf is also its outer child in T^*. Otherwise, outer$^*(v)$ = inner(outer(v)). The outer child of a shrink node is never a split node, and therefore outer(v) is a shrink node. By the shrink-split property, the inner child of a shrink node is a split node, and therefore outer$^*(v)$ is a split node. In either case, it cannot be a shrink node. Each time a shrink node is expanded, we charge its processing to its outer child. No visited leaf

node nor split node will receive more than a constant number of charges in this way.

Having dispensed with the small expanded shrink nodes, to complete the proof, it suffices to bound the number of small expanded split nodes. First, we claim that any node of size less than $w\varepsilon/\sqrt{d}$ cannot be expanded. For a node to be expanded its cell must intersect both the inner range Q^- and the complement of the outer range Q^+. Recall that the boundaries of Q^- and Q^+ are separated from each other by a distance of at least $w\varepsilon$. Since the diameter of a cell of size s is at most $s\sqrt{d}$, a cell of size less than $w\varepsilon/\sqrt{d}$ cannot intersect both range boundaries and hence cannot be expanded.

Thus, it remains to count the number of expanded split nodes of sizes from $2w$ down to $2w\varepsilon/\sqrt{d}$. As in the analysis of Theorem 1 of [12], we partition these nodes into groups according to their sizes. For $i \geq 0$, define *size group* i to be the set of nodes whose cell size is $1/2^i$. Since small nodes are of size less than $2w$, the first size group of interest is $a+1$, where $1/2^{a+1} < 2w \leq 1/2^a$, and hence $a = \lfloor -\lg 2w \rfloor$. Since nodes that are smaller than $2w\varepsilon/\sqrt{d}$ are not expanded, the last size group of interest is b, where $1/2^{b+1} < 2w\varepsilon/\sqrt{d} \leq 1/2^b$, and hence $b = \left\lfloor \lg \left(2w\varepsilon/\sqrt{d}\right) \right\rfloor$.

Because a node and its child may have the same size, we cannot apply the packing lemma directly to each size group. Define the *base group* for the ith size group to be the subset of nodes in the size group that are leaves or whose children are both in the next smaller size group. The cells corresponding to the nodes in a base group have pairwise disjoint interiors, since none of their descendants can be in the same base group. By the convexity of Q and a standard packing argument (e.g., Lemma 3 of [12]), it follows that the number of nodes in the ith base group is

at most

$$2d^2\left(1+\left\lceil\frac{2w}{1/2^i}\right\rceil\right)^{d-1} \;=\; 2d^2\left(1+\left\lceil w2^{i+1}\right\rceil\right)^{d-1}.$$

Since each successive split halves one of the side lengths of the outer box, the size of the nodes decrease by a factor 1/2 after any d consecutive splits by recursive invocation of the range search procedure. Thus at most $2d$ levels of split ancestors above the base group can be in the same size group, and thus the number of nodes in any size group is at most $2d$ times the above quantity.

Thus, over all of the base groups, the total number of expanded nodes, which we denote by N, is

$$N \;\le\; \sum_{i=a+1}^{b} 2d^2\left(1+\left\lceil w2^{i+1}\right\rceil\right)^{d-1}.$$

Observe that for $i \ge a+1$, we have $w2^{i+1} \ge w2^{a+2} \ge 1$. For any $x \ge 1$, note that $1+\lceil x\rceil \le 3x$, and hence we have

$$N \;\le\; \sum_{i=a+1}^{b} 2d^2\left(3w2^{i+1}\right)^{d-1} \;\le\; 2d^2\left(6w\right)^{d-1}\sum_{i=0}^{b}\left(2^{d-1}\right)^{i}.$$

Solving this geometric series yields

$$\begin{aligned} N &\le 2d^2(6w)^{d-1}\frac{(2^{d-1})^{b+1}-1}{2^{d-1}-1} \\ &\le 2d^2(6w)^{d-1}\frac{2^{d-1}\left(\sqrt{d}/\left(2w\varepsilon\right)\right)^{d-1}}{2^{d-1}-1} \\ &= 2d^2\frac{2^{d-1}}{2^{d-1}-1}\left(\frac{3\sqrt{d}}{\varepsilon}\right)^{d-1} \\ &\le 4d^2\left(\frac{3\sqrt{d}}{\varepsilon}\right)^{d-1}. \end{aligned}$$

Thus, the total number of small expanded nodes is bounded by $O((1/\varepsilon)^{d-1})$. Combining this with the $O(h)$ bound on the number of large expanded nodes completes the proof. $\square$

Chapter 4

A Self-adjusting Data Structure for Multidimensional Point Sets[1]

4.1 Introduction

In the previous chapter we presented a quadtree variant that uses a local restructuring operation, called promotion, to maintain its balance. This resulted in the quadtreap data structure, which (with high probability) can support insertion and deletion in $O(\log n)$ worst-case time. In some applications, especially when the probability distribution of accesses is highly skewed and varies over time, it is desirable that data structures adapt to achieve optimal performance for this access pattern. This leads to the concept of a *self-adjusting data structure*, whose structure adapts to the pattern of accesses over time. The best known example of such a structure for 1-dimensional data is the splay tree of Sleator and Tarjan [99].

In this chapter we present a multidimensional generalization of this structure, which we call the *splay quadtree*. Similar to the traditional splay tree, accesses are based on an splaying operation that restructures the tree in order to bring an arbitrary internal node to the root of the tree. We show that, given a splay quadtree with n points, the amortized time to splay a node is $O(\log n)$ (see Theorem 4.3.5),

[1]The material appearing in the chapter is based on the paper "A Self-adjusting Data Structure for Multidimensional Point Sets" [87].

and the amortized time to insert or delete a point is also $O(\log n)$ (see Theorems 4.4.1 and 4.4.2). We also present multidimensional generalizations of many of the results of [99], including the Balance Theorem, the Static Optimality Theorem, the Working Set theorem, and variants of the Static Finger Theorem for a number of different proximity queries, including box queries (which generalize point-location queries), approximate nearest neighbor queries, and approximate range counting queries.

The rest of the chapter is organized as follows. In Section 4.2, we present some background material on the BD tree, the variant of the quadtree on which the splay quadtree is based. In Section 4.3, we present the splaying operation and analyze its amortized time. In Section 4.4, we present and analyze algorithms for point insertion and deletion. Finally, in Section 4.5, we present the various search and optimality results.

4.2 Preliminaries

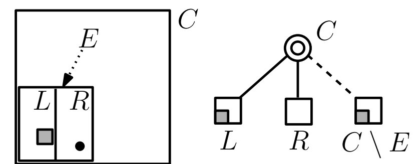
We begin by recalling some of the basic elements of quadtrees and BD-trees. A *quadtree* is a hierarchical decomposition of space into d-dimensional hypercubes, called *cells*. The root of the quadtree is associated with a unit hypercube $[0, 1]^d$, and we assume that (through an appropriate scaling) all the points fit within this hypercube. Each internal node has 2^d children corresponding to a subdivision of its cell into 2^d disjoint hypercubes, each having half the side length.

It will be convenient to consider a binary variant of the quadtree as bintrees [74, 95]. Each decomposition step splits a cell by an axis-orthogonal hyperplane that

bisects the cell's longest side. If there are ties, the side with the smallest coordinate index is selected. Although cells are not hypercubes, each cell has constant aspect ratio. Henceforth, we use the term *quadtree box* in this binary context.

If the point distribution is highly skewed, a compressed quadtree may have height $\Theta(n)$. One way to deal with this is to introduce a partitioning mechanism that allows the algorithm to "zoom" into regions of dense concentration. In [12,13] a subdivision operation, called *shrinking*, was proposed to achieve this. The resulting data structure is called a *box-decomposition tree* (*BD-tree*). (In fact, it would be more accurate to call our data structure a "splay BD-tree.") This is a binary tree in which the cell associated with each node is either a quadtree box or the set theoretic difference of two nested boxes, called the *outer box* and *inner box*. Although cells are not convex, they have bounded aspect ratio and are of constant combinatorial complexity.

The splay quadtree is based on a decomposition method that combines shrinking and splitting at each step. We say that a cell of a BD-tree is *crowded* if it contains two or more points or if it contains an inner box and at least one point. Let B denote this cell's outer box, and let C be any quadtree box contained within B, such that, if the cell has an inner box, C contains it. We generate a cell $O = B \setminus C$. We then split C by bisecting its longest side forming two cells L and R. By basic properties of quadtree boxes, the inner box (if it exists) lies entirely inside of one of the two cells. We make the convention, called the *inner-left convention*, that the inner box (if it exists) lies



within L. (Thus, the left child's cell is not necessarily geometrically to the left of the right child.) We generate a node with three children called the *left*, *right*, and *outer* child, corresponding to the cells L, R, and O, respectively. We allow for the possibility that $C = B$, which implies that the outer child's cell is empty, which we call a *vacuous leaf*. Our algorithms will maintain the invariant that the left and right children separate at least two entities (either two points or a point and inner box). It follows from the analysis given in [80] that the size of the tree is linear in the number of points.

As shown in [80], the tree can be restructured through a local operation, called *promotion*, which is analogous to rotation in binary trees. The operation is given any non-root internal node x and is denoted by promote(x). Let y denote x's parent. There are three cases depending on which child x is (see Fig. 4.1). Note that promotion does not alter the underlying subdivision of space.

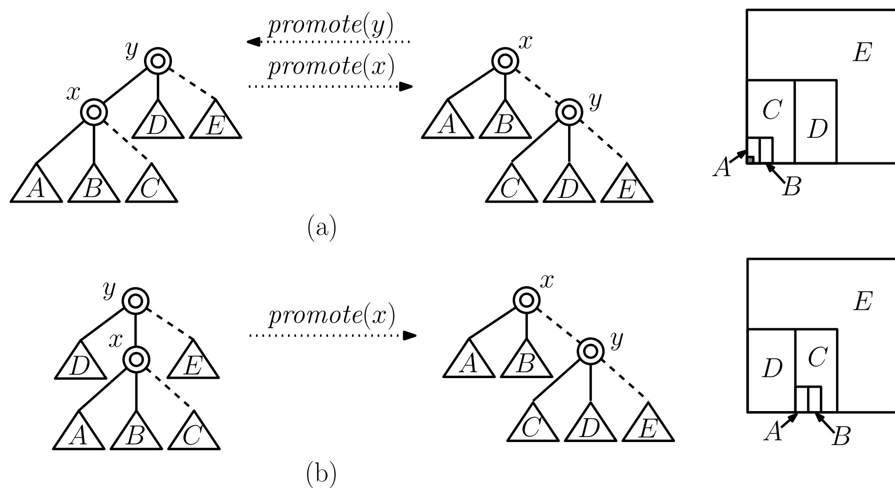


Fig. 4.1: (a) Left and outer promotions and (b) right promotion.

Left child: This makes y the outer child of x, and x's old outer child becomes the

new left child of y (see Fig. 4.1(a)). Observe that the inner-left convention is maintained, since x's left child is unchanged, and y's new left child (labeled C in Fig. 4.1(a)) contains the inner box (consisting of the union of A and B's cells in Fig. 4.1(a)).

Outer child: This is the inverse of the left-child promotion, where the roles of x and y are reversed (see Fig. 4.1(a)). In particular, let y be an internal node, and let x be y's parent. Then, promote(y) makes x the left child of y, and y's old left child becomes the new outer child of x. Again, the inner-left convention is maintained, since prior to the promotion, if x's cell had an inner box, it must lie in its left child (A in Fig. 4.1(a)), and after promotion, it lies in y's left child, x.

Right child: To maintain the convention that a cell has only one inner box, this operation is defined only if x's left sibling (labeled D in Fig. 4.1(b)) does not have an inner box. Letting y denote x's parent, promote(x) swaps x with its left sibling and then performs promote(x) (which is now a left-child promotion). Before the promotion, the inner box that contains x's shrinking box (consisting of the union of A and B's cells in Fig. 4.1(b)) is an inner box of x's outer child (labeled C in the figure). After promotion, this inner box lies within the left child of y, thus satisfying the inner-left convention.

It is easily verified that promotion can be performed in $O(1)$ time, and as shown above, it preserves the inner-left convention. The complicating feature of promotion is that right promotions can be performed only if the node's left sibling does not

have an inner box. (Note that an inner box is generated with each shrink operation, and so each outer child's cell has an inner box and this inner box is inherited by all of its descendents on its left-child path.) The reason for not allowing such a promotion is that it would result in x's cell having two inner boxes. Arbitrary repetition of this operation could then result in nodes whose cells have an unbounded number of inner boxes. This would violate one of the principal features of the hierarchical subdivision, namely that all cells have constant combinatorial complexity. We say that a promotion is *admissible*, if it is either a left or outer promotion or if it is a right promotion in which the left child has no inner box.

4.3 Self-adjusting Quadtrees

Like the standard splay tree [99], our self-adjusting quadtree, or *splay quadtree*, maintains no balance information and is based on an operation, called *splaying*, that moves an arbitrary internal node to the root through a series of promotions. This operation is described in terms of primitives, called *basic splaying steps*.

4.3.1 Basic Splaying Steps

We begin with some definitions. Let x be an internal node. If x is not the root, let $p(x)$ denote its parent, and if $p(x)$ is not root, let $g(x)$ denote x's grandparent. Define $rel(p(x), x)$ to be one of the symbols L, R, or O depending on whether x is the left, right, or outer child of $p(x)$, respectively. Define $str(x)$ be the string of at most two characters over $\{L, R, O\}$ representing x's relationship to its parent and

grandparent, if they exist. Formally, if x is the root, then $str(x)$ is the empty string, which we denote by ε. If $p(x)$ is the root of the tree, then $str(x)$ is the single character sequence $rel(p(x), x)$. Otherwise, $str(x)$ is defined to be two-character string whose first character is $rel(g(x), p(x))$ and whose second character is $rel(p(x), x)$. Given a non-root node x, a basic splaying step is defined as follows.

Basic Splay(x):

zig: If $p(x)$ is the root, perform promote(x) (see Fig. 4.2(a)).

zig-zag: If $str(x) \in \{LO, RO, OL\}$, do promote($x$), and then promote($x$) (see Fig. 4.2(b) for the case $str(x) = LO$).

zig-zig: Otherwise (if $str(x) \in \{LL, LR, RL, RR, OR, OO\}$), do promote($p(x)$) and then promote($x$) (see Fig. 4.2(c) for the case $str(x) = LL$).

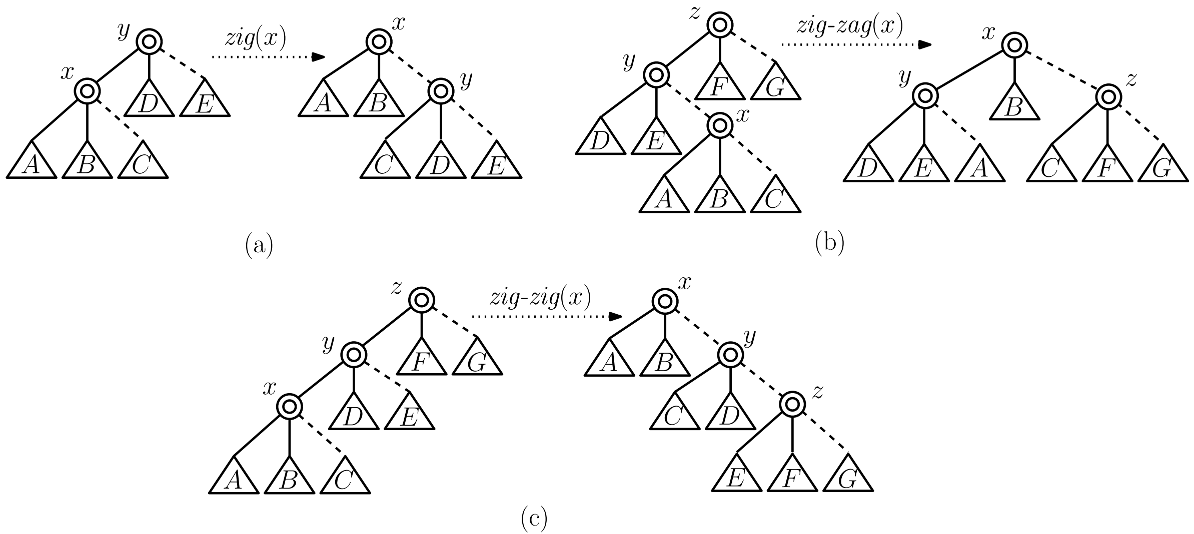


Fig. 4.2: The basic splaying operations: (a) *zig*, (b) *zig-zag*, and (c) *zig-zig*, where (before promotion) $y = p(x)$ and $z = g(x)$.

The astute reader will observe a slight lack of symmetry in the definitions of *zig-zag* and *zig-zig*. With the exception of OR, the *zig-zig* cases are of the form OO or ab, where $a, b \in \{L, R\}$, and the *zig-zag* cases are of the form aO or Ob. The principal reason for defining OR to be *zig-zig* is because the *zig-zag* processing would result in an inadmissible promotion. We say that a basic splaying step is *admissible* if all of its promotions are admissible. As we will see below, the admissibility of a basic splaying step at some node generally depends on the structure of the tree above the node.

Let us now analyze the amortized time of basic splaying. Subramanian [103] presented a generic analysis of splaying, and a slightly weaker form of our results could be derived from his analysis. We provide details for the sake of completeness. Following Sleator and Tarjan [99], we will employ a potential-based argument. We assume that each node x has been assigned a nonnegative weight $w(x)$. (Node weights will generally depend on the application.) Given such a weight assignment, define x's *size*, denoted $s(x)$, to be the sum of the weights of its descendants, including x itself. Define the *rank*, denoted $r(x)$, to be $\log s(x)$. (Unless otherwise specified, all logs are taken base 2.) Define the *potential* of the tree, denoted Φ, to be the sum of the ranks of all its nodes. The *amortized time* of an operation is defined to be $t + \Phi' - \Phi$, where t is the number of promotions performed, and Φ and Φ' are the potentials before and after the operation, respectively.

Consider a node x to which a basic splaying step is performed. Let $r(x)$ and $r'(x)$ be x's rank before and after the operation, respectively.

Lemma 4.3.1 *Given a splay quadtree T and any assignment of weights to its nodes, the amortized time for a zig operation at node x is at most $r'(x) - r(x) + 1$, and the amortized time for a zig-zag or zig-zig operation is at most $3(r'(x) - r(x))$.*

Proof: Let $y = p(x)$, and (if it exists) let $z = g(x)$ before the operation (see Fig. 4.2). Let $s(x)$ and $s'(x)$ denote x's size before and after the operation, respectively.

zig: One promotion is performed, and as a result, y's size can only decrease. So the amortized time is

$$\begin{aligned} a_{zig} &= 1 + r'(x) + r'(y) - r(x) - r(y) \\ &\leq 1 + r'(x) - r(x) \qquad \text{(since } r'(y) \leq r(y)\text{).} \end{aligned}$$

zig-zag: Two promotions are performed. Observe that x inherits all of z's former descendents, so $r'(x) = r(z)$. Since x is a child of y before the operation, $r(x) \leq r(y)$. So the amortized time is

$$\begin{aligned} a_{\textit{zig-zag}} &= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &= 2 + r'(y) + r'(z) - r(x) - r(y) \qquad \text{(since } r'(x) = r(z)\text{)} \\ &\leq 2 + r'(y) + r'(z) - 2r(x) \qquad \text{(since } r(x) \leq r(y)\text{).} \end{aligned}$$

We claim that this last sum is at most $2(r'(x) - r(x))$, that is, $r'(y) + r'(z) - 2r'(x) \leq -2$. To see this, observe that by the definition of the rank we have

$$r'(y) - r'(x) + r'(z) - r'(x) = \log \frac{s'(y)}{s'(x)} + \log \frac{s'(z)}{s'(x)} = \log \left(\frac{s'(y)}{s'(x)} \cdot \frac{s'(z)}{s'(x)} \right).$$

Since after the operation, y and z are distinct children of x, we have

$$\frac{s'(y)}{s'(x)} + \frac{s'(z)}{s'(x)} \leq 1.$$

Using the fact that the geometric mean does not exceed the arithmetic mean, we have

$$\begin{aligned}
\log\left(\frac{s'(y)}{s'(x)} \cdot \frac{s'(z)}{s'(x)}\right) &= 2\log\sqrt{\frac{s'(y)}{s'(x)} \cdot \frac{s'(z)}{s'(x)}} \\
&\leq 2\log\left(\frac{1}{2}\left(\frac{s'(y)}{s'(x)} + \frac{s'(z)}{s'(x)}\right)\right) \\
&\leq 2\log\frac{1}{2} \\
&= -2,
\end{aligned}$$

as desired. Thus, the amortized time of *zig-zag* is at most $2(r'(x) - r(x)) \leq 3(r'(x) - r(x))$.

zig-zig: Two promotions are performed. As before, $r'(x) = r(z)$ and $r(x) \leq r(y)$. After promotion, y is a child of x, so $r'(y) \leq r'(x)$. So the amortized time is

$$\begin{aligned}
a_{\textit{zig-zig}} &= 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
&= 2 + r'(y) + r'(z) - r(x) - r(y) && (\text{since } r'(x) = r(z)) \\
&\leq 2 + r'(y) + r'(z) - 2r(x) && (\text{since } r(x) \leq r(y)) \\
&\leq 2 + r'(x) + r'(z) - 2r(x) && (\text{since } r'(y) \leq r'(x)).
\end{aligned}$$

We claim that this last sum is at most $3(r'(x) - r(x))$, that is, $r(x) + r'(z) -$

$2r'(x) \leq -2$. As before, observe that by the definition of the rank we have

$$r(x) - r'(x) + r'(z) - r'(x) \;=\; \log \frac{s(x)}{s'(x)} + \log \frac{s'(z)}{s'(x)} \;=\; \log \left(\frac{s(x)}{s'(x)} \cdot \frac{s'(z)}{s'(x)} \right).$$

It is easy to verify that x's subtree before the operation and z's subtree after the operation are disjoint. (The cases of the form xy for $x, y \in \{L, R\}$ are structurally similar to Fig. 4.2(c), and the case OO is just the reverse. The OR case is illustrated in Fig. 4.4 below.) Thus, we have

$$\frac{s(x)}{s'(x)} + \frac{s'(z)}{s'(x)} \;\leq\; 1.$$

Using the fact that the geometric mean does not exceed the arithmetic mean, we have

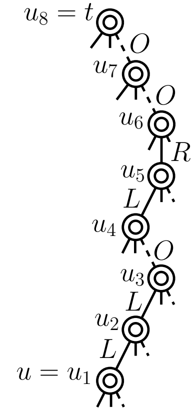
$$\begin{aligned}
\log \left(\frac{s(x)}{s'(x)} \cdot \frac{s'(z)}{s'(x)} \right) &= 2 \log \sqrt{\frac{s(x)}{s'(x)} \cdot \frac{s'(z)}{s'(x)}} \\
&\leq 2 \log \left(\frac{1}{2} \left(\frac{s(x)}{s'(x)} + \frac{s'(z)}{s'(x)} \right) \right) \\
&\leq 2 \log \frac{1}{2} \\
&= -2,
\end{aligned}$$

as desired. Thus, the amortized time of *zig-zig* is at most $3(r'(x) - r(x))$.

$\square$

4.3.2 Splaying Operation

As mentioned earlier, the splay quadtree is based on an operation that modifies the tree so that a given internal node becomes the root. We start with a definition. Let T be a splay quadtree, let u be a node of T, and let t be T's root. Letting $\langle u = u_1, u_2, \ldots, u_k = t \rangle$ denote the path from u to t, the *path sequence* $seq_T(u)$ is defined



$$seq_T(u) \;=\; rel(u_k, u_{k-1}) \ldots rel(u_3, u_2) rel(u_2, u_1).$$

(Note that we list paths bottom-up but the sequence string is given top-down. For example, in the figure to the right we have $seq_T(u) = OORLOLL$.) Using the terminology of regular expressions, this sequence can be thought of as a string in the language $\{L, R, O\}^*$. In the standard Sleator-Tarjan binary splay tree, basic splaying steps are performed bottom-up along the entire access path. If we were to ignore admissibility, we can define an analogous approach for splaying along an entire path. The code is presented in Algorithm 4. It is easy to see that the resulting sequence of basic splaying steps results in u becoming the new root of the tree (see Fig. 4.3).

Algorithm 4 Simple-splaying(T, u)

```
1: x ← u
2: while x ≠ root(T) do
3:     if p(x) = root(T) then zig(x)
4:     else if str(x) ∈ {LO, RO, OL} then zig-zag(x)
5:     else zig-zig(x)
```

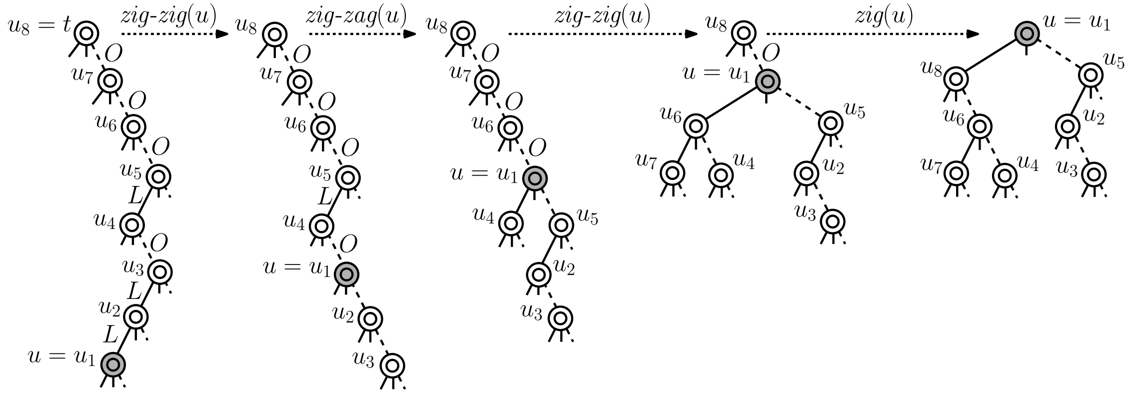


Fig. 4.3: Applying simple splaying along a path with $seq_T(u) = OOOLOLL$.

Of course, we wish to avoid inadmissible promotions. Recall that a promotion is inadmissible whenever a right child is being promoted, and its left sibling has an inner box. As we descend the tree, new inner boxes are generated only when we visit an outer child. By the inner-left convention this box is then inherited by all the nodes along its chain of left-child descendents. It follows that a promotion is admissible at a node u provided that $seq_T(u)$ does not end with a substring of the form OL^*R. Observe that after we apply a basic splaying step at u, $seq_T(u)$ is transformed by the removal one one or two symbols from its right end (depending on whether the operation is a *zig* or *zig-zig*/*zig-zag*). This means that if $seq_T(u)$ has no substring of this form, then after repeated applications of basic splaying steps, it cannot end in a substring of this form. As a consequence we have the following useful observation.

Lemma 4.3.2 *Given a node u in a splay quadtree T, if $seq_T(u)$ does not contain a substring of the form OL^*R, then the promotions generated by Simple-splaying(T, u) are all admissible.*

Our splaying algorithm for the general case involves a three-phase approach.

The first two phases involve basic splaying steps in order to transform the tree so that $seq_T(u)$ satisfies the conditions of Lemma 4.3.2. In particular, the first phase transforms the path sequence by compressing all subsequences consisting of two or more Os or Ls into a subsequence consisting of at most one such character. The second phase modifies the path such that the left sibling of any right child on the path does not have an inner box. We will see that this can be done by eliminating all OR subsequences. After this, the third phase simply invokes Simple-splaying(T, u). The complete procedure, which we call Splay(T, u) is presented below.

Phase 1: Any string over $\{O, L, R\}^*$ can be viewed as a sequence of substrings of the form $\{O, L\}^*$ separated by Rs. Thus, $seq_T(u) \in \{\{O, L\}^* R\}^* \{O, L\}^*$. Letting x denote the current node, we repeatedly applying basic splaying steps whenever $str(x) \in \{LL, OO, LO, OL\}$. Since each application eliminates the last two symbols from the sequence, this eventually compresses each $\{O, L\}^*$ subsequence to one of the form $\{O, L, \varepsilon\}$. Thus, the output sequence consists of a series of Rs optionally interspersed with a single O or L. That is, it has the form $\{\{O, L, \varepsilon\}R\}^* \{O, L, \varepsilon\}$.

This phase is shown in Algorithm 5. Lines 3–5 of the procedure perform the basic splaying operations, and lines 6–8 jump to the next $\{O, L\}^*$ subsequence. On exiting the inner while loop, we know that if neither x nor its parent is the root, then one of these nodes is a right child. Lines 6–8 advance x by jumping over this R link, thus moving x to the next $\{O, L\}^*$ subsequence, from which the process is repeated. Because no right children are involved, all the basic

splaying operations are admissible.

Algorithm 5 Splaying-Phase-1(T, u)

```
1: x ← u
2: while x ≠ root(T) do
3:     while str(x) ∈ {LL, OO, LO, OL} do
4:         if str(x) ∈ {LL, OO} then zig-zig(x)
5:         else zig-zag(x)
6:     if x = root(T) or p(x) = root(T) then return
7:     else if rel(p(x), x) = R then x ← p(x)
8:     else x ← g(x)                              ▷ rel(g(x), p(x)) = R
```

Phase 2: In this phase, we transform $seq_T(u)$ such that the sibling node of any right child on the path does not have an inner box. To motivate the processing of this phase, recall that our objective is to eliminate substrings of the form OL^*R. After the first phase $seq_T(u) \in \{\{O, L, \varepsilon\}R\}^*\{O, L, \varepsilon\}$. In such a string, there is an R between each occurrence of an O and L, and so the only way a substring of the form OL^*R could arise is as OR.

Consider a node x where $str(x) = OR$ (see Fig. 4.4(a)). Since x's parent is an outer child, x's left sibling (labeled C in Fig. 4.4(a)) has an inner box. Let us consider the effect of performing *zig-zig*(x). Let y and z denote x's parent and grandparent, respectively. By the preconditions of this phase, $seq_T(u)$ has no LO or OO substrings. Therefore, z is either the root or a right child. In either case, its cell has no inner box. First, we perform promote(y) (see Fig. 4.4(b)). After promote(y), x's new left sibling, z, does not have an inner box (see Fig. 4.4(b)). Therefore, we may now safely perform promote(x) (see Fig. 4.4(c)). (Note that this operation is the essential reason for the lack of symmetry that we noted when presenting the basic splaying steps.)

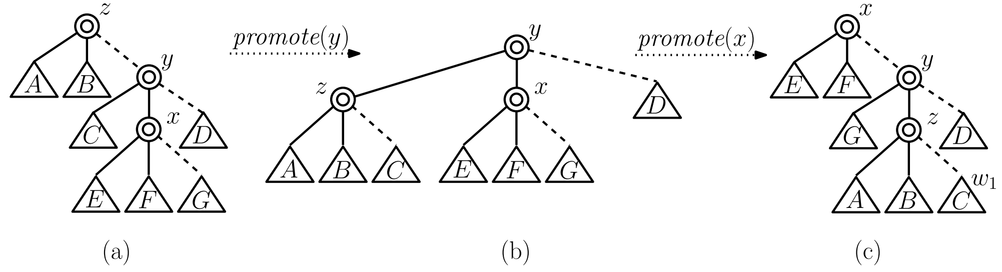


Fig. 4.4: A *zig-zig* splaying when $str(x) = OR$.

Observe that x's old outer child (labeled G in Fig. 4.4(a)) becomes the left child of x's new outer child, y after the *zig-zig* operation. Thus, if the original splaying node u was x's outer child, the value of $str(u)$ would be mapped from ORO to OL. In general, an initial path subsequence consisting of consecutive OR alternations will be mapped to the subsequence OL^*. Therefore, after removing all the OR substrings, the initial path sequence will be transformed to a sequence of the form $\{LR, R\}^*\{OL^*, O, L, \varepsilon\}$. This phase is shown in Algorithm 6.

Algorithm 6 Splaying-Phase-2(T, u)

1: $x \leftarrow u$
2: **while** $x \neq \text{root}(T)$ **do**
3:     **if** $str(x) = OR$ **then** *zig-zig*(x)
4:     **else** $x \leftarrow p(x)$

Phase 3: After Phase 2 we have $seq_T(u) \in \{LR, R\}^*\{OL^*, O, L, \varepsilon\}$. In such a string all the occurrences of O come after all the occurrences of R, which implies that it cannot contain the substring OL^*R. Therefore, by Lemma 4.3.2 we may apply Simple-splaying(T, u) to finish up the splaying process, thus moving u to the root of the tree.

Next, let us analyze the amortized time of our splaying algorithm. Our analysis

is a straightforward generalization of the analysis given by Sleator and Tarjan [99]. The added wrinkle is that we do not always apply a basic splaying operation at each node along the access path, but rather we may skip over some nodes. Define an *admissible splaying sequence* to be a sequence of non-root nodes $\langle x_1, \ldots, x_k \rangle$, such that if we apply basic splaying operations at each node of the sequence (in order), each splaying operation is admissible, and for $1 \leq i \leq k-1$, after the first i operations, node x_{i+1} is either equal to or an ancestor of x_i. (Note that the same node may appear many times in such a sequence.) It is easy to verify that Algorithms 4–6 generate only admissible splaying sequences. (We have already demonstrated admissibility when we presented the algorithms. The ancestor condition follows because whenever a phase skips over a node, it always moves to an ancestor.) The following lemma establishes the amortized time for each phase.

Lemma 4.3.3 *Given a splay quadtree T with root t and any assignment of weights to its nodes, let $\langle x_1, \ldots, x_k \rangle$ be an admissible splaying sequence in T. If no zig operation is performed, the total amortized time to perform basic splaying steps along the nodes of this subsequence is at most $3(r(t) - r(x_1))$. If there is a zig operation, the total amortized time is at most $3(r(t) - r(x_1)) + 1$. (All ranks are defined for the tree before executing the operations.)*

Proof: Recall that $r(x_i)$ and $r'(x_i)$ denote node x_i's rank before and after performing a basic splaying step, respectively. By Lemma 4.3.1, the amortized time to perform a *zig* operation at node x_i is at most $r'(x_i) - r(x_i) + 1$, and the amortized time for a single *zig-zag* or *zig-zig* is at most $3(r'(x_i) - r(x_i))$. Define x_{k+1} to be the

root of the tree after the last operation. By admissibility, for $1 \leq i \leq k$, x_{i+1} is either equal to or an ancestor of x_i. Therefore, for $1 \leq i \leq k$, we have $r'(x_i) \leq r(x_{i+1})$. Thus, ignoring the additional $+1$ term for a possible *zig* operation, which can be applied at most once, the total amortized time for all basic splaying steps at nodes x_i $(1 \leq i \leq k)$ is at most

$$\sum_{i=1}^{k} 3(r'(x_i) - r(x_i)) \;\leq\; 3\sum_{i=1}^{k}(r(x_{i+1}) - r(x_i)) \;=\; 3(r(t) - r(x_1)),$$

as desired. □

Observe that Phases 1 and 2 involve only *zig-zig* and *zig-zag* operations. By combining the amortized times for each of the three phases, and observing that a *zig* can be performed at most once at the end of Phase 3, we obtain the following. (As in Lemma 4.3.3 ranks and sizes are defined for the tree before executing the operations.)

Lemma 4.3.4 *Given a splay quadtree T with root t and any assignment of weights to its nodes, the amortized time to perform Splay(T, u) is at most $9(r(t)-r(u))+1 = O(1 + \log(s(t)/s(u)))$.*

We can now present the main result of this section.

Lemma 4.3.5 *Given a splay quadtree T containing n points and any internal node u of T, the amortized time to perform Splay(T, u) is $O(\log n)$.*

Proof: Consider the following weight assignment $w(x)$ to the nodes x of the tree. Each leaf node that contains a point is assigned a weight of $1/n$, and the remaining

nodes of the tree (internal nodes and leaves containing inner boxes) are assigned a weight of zero. Recall that the size $s(x)$ of any node x is defined to be the sum of the weights of x and all its descendants, which is just the number of points contained in the corresponding subtree. Under this weight assignment the root node t has a size of $s(t) = 1$. Since the subtree rooted at any internal node contains at least one point, we have $s(u) \geq 1/n$. By Lemma 4.3.4, the amortized time of Splay(T, q) is $O(\log(s(t)/s(u))) = O(\log n)$, as desired. $\square$

In traditional splay trees keys are stored in the internal nodes, and splaying on a node brings its key to the root. This is a central reason behind the self-adjusting nature of splay trees. As we mentioned earlier, the operation Splay(T, u) assumes that u is an internal node of the tree, and it has the effect of bringing u to the root of the tree. But the points of the tree are not stored in internal nodes, they are stored in the leaves. This raises the question of whether there is a corresponding notion of bringing a point's leaf node near to the root of the tree. Ideally, we would like to claim that splaying on the parent of a leaf node will bring the leaf node up so it becomes a child of the tree's root. This is not generally true, but our next lemma shows that it is true if the leaf node contains a point (as opposed to containing an inner box). Further, the relationship between this leaf and its parent is unchanged after splaying.

Lemma 4.3.6 *Let u be a leaf node of the splay quadtree that contains a point. After applying Splay$(T, p(u))$, $p(u)$ is mapped to the root of the tree, u remains a child of $p(u)$, and u's relationship to $p(u)$ is unchanged.*

Proof: We assert that whenever $p(u)$ is promoted, u remains the same child of $p(u)$. To establish this, observe that since u contains a point, it cannot contain an inner box. This implies that u cannot be an outer child, and it cannot be the left child of an outer child. Consider the possible result of any promotion being applied to $p(u)$ (recall Fig. 4.1). In the case of a left-child or right-child promotion, both the left and right children (labeled A and B in Fig. 4.1(a) and (b)) remain the same children of their respective parents. In the case of an outer-child promotion, we need only consider the right child. The right child (labeled D in Fig. 4.1(a), read from right to left) remains the right child of its parent after promotion.

This establishes that u remains the same child of $p(u)$ after a single promotion. Because the effect of Splay$(T, p(u))$ is to bring $p(u)$ to the root through a series of promotions, the lemma follows by simple induction. □

Before concluding this section, we add a final observation. When we discuss some of our applications below, it will be useful to invoke the splaying operation on subtrees of T. We allow the first argument of Splay(T, u) to be an arbitrary subtree T' of an existing splay tree. This is effectively equivalent to detaching T' (by unlinking it from its parent), applying Splay(T', u) to the detached tree, and finally reattaching the result back at its original position.

4.4 Insertion and Deletion

In this section we consider how to insert and delete points from the splay quadtree. Let us first consider the insertion of a new point q. We begin by inserting the point

into the tree following the incremental construction given in [80]. In particular, the leaf node containing q is located by a straightforward descent of the tree. Let C denote the outer box of this leaf's cell, and let b denote its contents, which is either a point or an inner box (see Fig. 4.5). We create a new internal node u to replace this leaf as follows. Let E denote the smallest quadtree box that contains both q and b. (Clearly, E is either contained within or is equal to C.) We create a leaf node whose outer box is C and whose inner box is E and make this u's outer child. Next, we then split E, which by the minimality of E, separates b from q. We generate two leaf nodes, one for each side of the split. We make the node containing b the left child of u and the node containing q the right child. (Since b may be an inner box, this satisfies the inner-left convention.) After creating u, we invoke Splay(T, u), which brings u to the root of the tree. By Lemma 4.3.6, q's leaf node is now the right child of the tree's root.

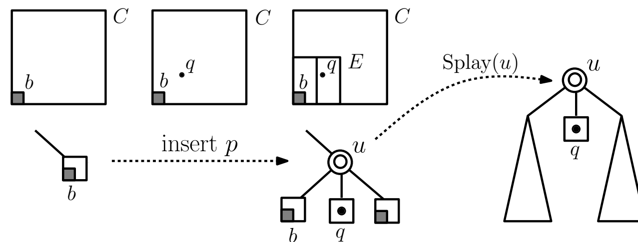


Fig. 4.5: Point insertion.

To analyze the amortized time of this operation, observe that the time needed to locate the leaf node containing q is proportional to u's depth in the tree. Therefore, the time to do this is dominated by the time to perform Splay(T, u). The time needed to create u and its children and link them into the tree is $O(1)$. By Theorem 4.3.5, we have the following.

Theorem 4.4.1 *Given a splay quadtree T with n points, the insertion of a point into T can be performed in $O(\log n)$ amortized time.*

Next, let us consider the deletion of an existing point q. As with the deletion of points from a standard binary search tree, deletion is more complex than insertion. It is trivial to delete a point that has just been inserted, by simply undoing to the insertion process. However, subsequent insertions result in changes to the tree's structure, and this complicates the removal process. Our approach will be to apply splaying operations so that the tree is structured *as if* the point q had been the last point to be inserted.

We first determine the leaf node v containing the deleted point. Let $T_{p(v)}$ be the subtree rooted at $p(v)$. We then restructure $T_{p(v)}$ by a method that will be described below. After restructuring, we shall see that $T_{p(v)}$ will be transformed into a subtree consisting of a single internal node whose left and outer children are leaf nodes and whose right child is a leaf node containing q. This is exactly the structure we would have if q had been the last point to be inserted. We complete the deletion process by effectively "undoing" the insertion process, as described above. In particular, we will replace $T_{p(v)}$ with a single leaf node (see [80] for further details.) We then splay the parent node of this replaced leaf node.

All that remains is to give the restructuring process, which is presented in Algorithm 7. Note that v cannot be an outer child, for otherwise it would have an inner box, rather than a point. If v is a left child, then neither it nor its right sibling has an inner box, and there will be no violation of the inner-left convention if we

swap v with its right child. Therefore, we may assume that v is a right child. Let w_1 and w_2 denote v's left and outer siblings.

Algorithm 7 Restructuring the tree as part of the deletion of a point in a leaf v.

1: **if** $v = \text{left}(p(v))$ **then** swap v and its left sibling
2: $w'_1 \leftarrow w_1 \leftarrow \text{left}(p(v))$
3: **if** w'_1 is not a leaf **then**
4:     **while** $\text{outer}(w'_1)$ is not a leaf **do** $w'_1 \leftarrow \text{outer}(w'_1)$
5:     $\text{Splay}(T(p(v), w'_1)$
6: $w'_2 \leftarrow w_2 \leftarrow \text{outer}(p(v))$
7: **if** w'_2 is not a leaf **then**
8:     **while** $\text{left}(w'_2)$ is not a leaf **do** $w'_2 \leftarrow \text{left}(w'_2)$
9:     $\text{Splay}(T(p(v), w'_2)$

First, consider the chain of outer children starting at w_1 until reaching a leaf node (see Fig. 4.6(a)). Let l_1 denote this leaf and w'_1 be its parent. In Step 5 of Algorithm 7 we splay w'_1 in the subtree $T_{p(v)}$. The path sequence of w'_1 in the subtree $T_{p(v)}$ is $seq_{T_{p(v)}}(w'_1) = LO^*$. It is easy to verify that the splaying operation at w'_1 involves a sequence of outer-child promotions (resulting from *zig-zig* splaying steps) followed by one last promotion of a left child. After the outer-child promotions, l_1 is still an outer child of w'_1 (see Fig. 4.6(b)). The final left-child promotion comes about either as a *zig* step (if the outer chain is of even length) or the second promotion of a *zig-zag* step (if the outer chain is of odd length). It brings w'_1 to the root of the subtree $T_{p(v)}$ and makes l_1 a left child of $p(v)$ (see Fig. 4.6(c)). After this, v's left sibling is a leaf node.

Symmetrically, consider the chain of left children starting at w_2 until reaching a leaf node (see Fig. 4.7(a)). Let l_2 denote this leaf and w'_2 be its parent. In Step 9 of Algorithm 7 we splay w'_2 in the subtree $T_{p(v)}$. The path sequence of w'_2 in the subtree

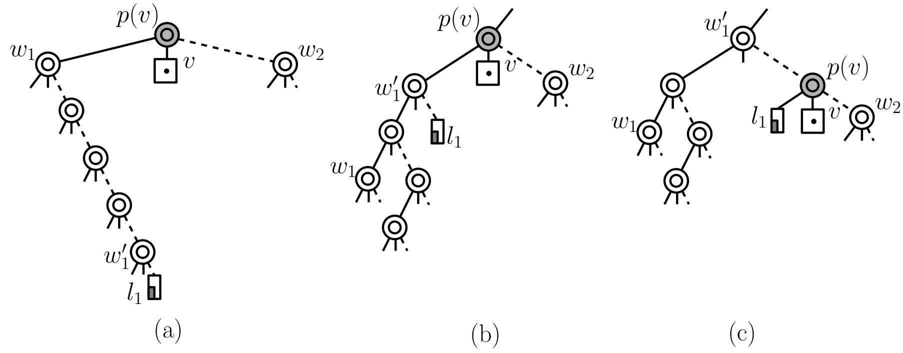


Fig. 4.6: Illustrating Steps 2–5 of Algorithm 7.

$T_{p(v)}$ is $seq_{T_{p(v)}}(w_2') = OL^*$. By symmetry, the splaying operation at w_2' involves a sequence of left-child promotions followed by one last promotion of an outer child. After the left-child promotions, l_2 is still a left child of w_2' (see Fig. 4.7(b)). The final outer-child promotion brings w_2' to the root of the subtree $T_{p(v)}$ and makes l_2 an outer child of $p(v)$ (see Fig. 4.7(c)). Thus, after splaying v's outer sibling is also a leaf node.

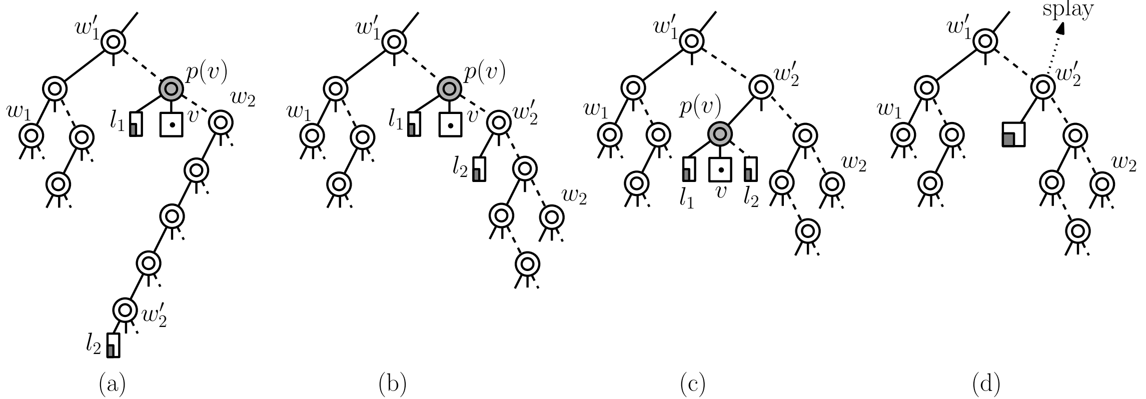


Fig. 4.7: Illustrating Steps 6–9 of Algorithm 7.

On completion of the restructuring process, v is a right child with two leaves as its left and outer siblings (see Fig. 4.7(c)). This is exactly the structure we desire (recall Fig. 4.5). As mentioned above, we can now locally "undo" the insertion of q and v (see Fig. 4.7(d)), and we splay the parent node of this replaced leaf node.

To analyze the amortized time of this operation, observe that the time needed to locate the leaf nodes l_1 and l_2 is dominated by the time to splay their respective parents. The time needed to replace v locally $O(1)$. Therefore, the overall deletion time is dominated by the time to perform the splays, which by Theorem 4.3.5 is $O(\log n)$.

Theorem 4.4.2 *Given a splay quadtree T with n points, the deletion of a point from T can be performed in $O(\log n)$ amortized time.*

4.5 Search and Optimality Results

In this section, we present theorems related to the complexity of performing various types of geometric searches on splay quadtrees. We provide generalizations to a number of optimality results for the standard splay tree, including the Balance Theorem, the Static Optimality Theorem, the Working Set theorem, and variants of the Static Finger Theorem. As in [99], our approach is based on applying Lemma 4.3.4 under an appropriate assignment of weights to the nodes.

Before presenting our results, we begin with a technical lemma that will be useful in the analyses of this section. Throughout this section, P denotes a set of n points in $\mathbb{R}^d$ (where d is a constant), T is a splay quadtree for P, I denotes the set of internal nodes of T, L denotes the set of leaf nodes, and Z denotes the subdivision induced by T's cells.

Lemma 4.5.1 *Let T be a splay quadtree, let t be its root, and consider any assignment of weights $0 \leq w(v) \leq 1$ to the nodes v of T. Given any sequence of valid*

splay operations on T, the total decrease in the potential over this sequence is at most $|I| \cdot r(t) - \sum_{l \in L} r(l)$.

Proof: Recall that the potential of the tree is the sum of the ranks of all its nodes. Note that the splaying operation does not change the rank of any leaf node since each splay acts only on internal nodes. Thus, only the ranks of internal nodes have an effect on the decrease in the potential.

Consider the rank $r(u)$, of an internal node u. Recall that $r(u)$ is the logarithm of the sum of weights of all descendants. In particular, if let $D(u)$ be the set of the leaves among u's descendants, $r(u)$ is at least $\log \sum_{v \in D(u)} w(v)$. Because the weights lie in the interval $[0, 1]$, we have

$$r(u) \geq \log \sum_{v \in D(u)} w(v) \geq \log \prod_{v \in D(u)} w(v) = \sum_{v \in D(u)} \log w(v).$$

The rank of any internal node is at most the rank $r(t)$ of the root t. Let $\Delta r(u)$ denote the change in rank of u over the entire sequences of splays. Therefore, the decrease in potential of the tree, which is denoted by $\Delta\Phi$, is

$$\begin{aligned} \Delta\Phi &= \sum_{u \in I} \Delta r(u) \leq \sum_{u \in I} \left(r(t) - \sum_{v \in D(u)} \log w(v) \right) \\ &= |I| \cdot r(t) - \sum_{u \in I} \sum_{v \in D(u)} \log w(v) \ \leq \ |I| \cdot r(t) - \sum_{l \in L} \log w(l). \end{aligned}$$

□

4.5.1 Basic Optimality Results

We first discuss theorems related to the search time for accessing points. These theorems are essentially the same as those proved in [99] for 1-dimensional splay trees. Given a query point q, an *access query* returns the leaf node of T whose cell contains q. (Thus, it can be thought of as a point location query for the subdivision of space induced by T.) An access query is processed by traversing the path from the root to the leaf node containing q, and splaying the parent of this leaf.

The working set property states that once an item is accessed, accesses to the same item in the near future are particularly efficient [99]. Through the application of an appropriate weight assignment, similar to one used in [99], and applying Lemma 4.5.1, we obtain the following.

Theorem 4.5.1 (Working Set Theorem) *Consider a set P of n points in $\mathbb{R}^d$. Let $q_1, \ldots, q_m$ be a sequence of access queries. For each access q_j $(1 \leq j \leq m)$, let t_j be the number of different cells accessed before q_j since its previous access, or since the beginning of the sequence if this is q_j's first access. The total time to answer m queries is $O(\sum_{j=1}^{m} \log(t_j + 1) + m + n \log n)$.*

Proof: We apply the reassignment of weights as in Sleator and Tarjan [99]. In particular, we assign the weights $1, 1/4, 1/9, \ldots, 1/|Z|^2$ to each cells in order by first access. The cell access earliest receives the largest weight. As each access occurs, redefine the weights as follow. Suppose the weight of cell z_j during access j is $1/k^2$. After access j, assign weight 1 to z_j, and for each cell z having a weight of $1/(k')^2$ with $k' < k$, assign it a weight of $1/(k'+1)^2$. Note that this reassignment permutes

the weights $1, 1/4, 1/9, \ldots, 1/|Z|^2$ among the cells, and makes the weight of cell z_j during access j to be $1/(t_j + 1)^2$.

We assign each leaf cell's weight to the corresponding leaf node and a weight of zero to each internal node. Let u_j be the parent internal node of the node containing leaf cell z_j $(1 \leq j \leq m)$. The size $s(u_j)$ of a node u_j is at least $w(z_j) = 1/(t_j + 1)^2$. The size $s(t)$ of the root t is

$$\sum_{z \in Z} w(z) = \sum_{k=1}^{|Z|} \frac{1}{k^2} < \sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6} < 2. \tag{4.1}$$

By Lemma 4.3.4, the amortized time to access j is at most

$$9(\log s(t) - \log s(u_j)) + 1 \leq 9 \left(\log 2 - \log \frac{1}{(t_j + 1)^2} \right) + 1 = 18 \log(t_j + 1) + 10.$$

The weight reassignment after an access increases the weight of z_j, whose corresponding leaf node is a child of the new root u_j, and only decreases the weights of other cells. The size of the root is unchanged, but the sizes of the other internal nodes can decrease. Thus, the amortized time during reassignment is either zero or negative. Therefore, the total amortized time is

$$\begin{aligned} \sum_{j=1}^{m} (9(\log s(t) - \log s(u_j)) + 1) &= \sum_{j=1}^{m} (18 \log(t_j + 1) + 10) \\ &= O\left(\sum_{j=1}^{m} \log(t_j + 1) + m \right). \end{aligned}$$

By Lemma 4.5.1, the decrease in potential is at most $|I| \cdot r(t) - \sum_{l \in L} w(l)$. By

our assignment, $\sum_{l \in L} \log w(l) = \sum_{z \in Z} \log w(z)$, and $w(z)$ is at least $1/|Z|^2$ by our assignment. Since the size of the tree is linear in n, the net decrease in potential of a tree over the m accesses is

$$\begin{aligned} \Delta\Phi \leq |I| \cdot r(t) - \sum_{l \in L} \log w(l) &\leq |I| \cdot \log 1 - \sum_{z \in Z} \log \frac{1}{|Z|^2} \\ &= O(|Z| \log |Z|) \;=\; O(n \log n). \end{aligned}$$

Thus, the actual total time is $O(\sum_{j=1}^{m} \log(t_j + 1) + m + n \log n)$. □

Two additional properties follow as consequences of the above theorem (see, e.g., Iacono [68]):

Theorem 4.5.2 (Balance Theorem) *Consider a set P of n points in $\mathbb{R}^d$ and a splay quadtree T for P. Let $q_1, \ldots, q_m$ be a sequence of access queries. The total time to answer these queries is $O((m + n) \log n + m)$.*

Theorem 4.5.3 (Static Optimality Theorem) *Given a subdivision Z, and the empirical probability p_z for each cell $z \in Z$, the total time to answer m access queries is $O(m \cdot entropy(Z))$.*

4.5.2 The Static Finger Results

In traditional splay trees, the *static finger theorem* states that the running times of a series of accesses can be bounded by the sum of logarithms of the distances of each accessed item to a given fixed key, called a *finger*. In the 1-dimensional context, the notion of distance is based on the number of keys that lie between two items [99].

Intuitively, this provides a measure of the degree of locality among a set of queries with respect to a static key. Generalizing this to a geometric setting is complicated by the issue of how to define an appropriate notion of distance based on *betweenness*. In this section, we present a number of static finger theorems for different queries in the splay quadtree.

In general, our notion of distance to a static finger is based on the number of relevant objects (points or quadtree cells) that are closer to the finger than the accessed object(s). Define the distance from a point p to a geometric object Q, denoted $\mathrm{dist}(p, Q)$, to be the minimum distance from p to any point of Q. Let $b(f, r)$ denote a ball of radius r centered at a point f. Our first static-finger result involves access queries.

Theorem 4.5.4 *Consider a set P of n points in $\mathbb{R}^d$, for some constant d. Let $q_1, \ldots, q_m$ be a sequence of access queries, where each q_j $(1 \leq j \leq m)$ is a point of P. If f is any fixed point in $\mathbb{R}^d$, the total time to access points $q_1, \ldots, q_m$ is $O\big(\sum_{j=1}^{m} \log N_f(q_j) + m + n \log n\big)$, where $N_f(q_j)$ is the number of points of P that are closer to f than q_j is.*

Proof: Since f is fixed throughout, let $N(p) = N_f(p)$. We assign a weight of $w(p) = 1/N(p)^2$ to each point $p \in P$. (This weight is used only to analyze the query time, and is not stored in the tree.) Each leaf node that has a point is assigned a weight equal to the point's weight, and otherwise its weight is zero. We assign each internal node a weight of zero. Recall that the size $s(x)$ of any node x is defined to be the sum of the weights of x and its descendants, and corresponds to total weight

of all points in the subtree rooted at x.

For the analysis of the amortized time, we first compute the size $s(t)$ of the root t. This is the sum of the weights of all points in P. If we let $p_1, \ldots, p_n$ be the sequence of points in nondecreasing distance order from f, the sequence of $N(p_1) \leq \cdots \leq N(p_n) = n$ is nondecreasing. Thus, by Equation (4.1) of Theorem 4.5.1, the total weight of all points is

$$\sum_{p \in P} w(p) = \sum_{i=1}^{n} w(p_i) = \sum_{i=1}^{n} \frac{1}{N(p_i)^2} \leq \sum_{i=1}^{n} \frac{1}{i^2} < 2. \tag{4.2}$$

Recall that accessing a point involves finding the leaf node containing this point and splaying its parent to the root. Let u_j be the parent node of the leaf node containing q_j. The size of node u_j is at least $w(q_j) = 1/N(q_j)^2$. Thus, the net amortized time of all m accesses is

$$\begin{aligned} \sum_{j=1}^{m} \big(9(\log s(t) - \log s(u_j)) + 1\big) &\leq \sum_{j=1}^{m} \left(9\left(\log 2 - \log \frac{1}{N(q_j)^2}\right) + 1\right) \\ &= O\left(\sum_{j=1}^{m} \log N(q_j) + m\right). \end{aligned}$$

Now consider the decrease in potential over all m accesses. By Lemma 4.5.1, it is at most $|I| \cdot r(t) - \sum_{l \in L} \log w(l)$. Since the size of the tree is linear in n, $|I| = O(n)$. By our assignment, $\sum_{l \in L} \log w(l) \leq \sum_{p \in P} w(p)$, and since $N(p) \leq |P|$, $w(p)$ is at

least $1/|P|^2$. Thus, the net decrease in potential of a tree over all m accesses is

$$\Delta\Phi \leq |I| \cdot r(t) - \sum_{l \in L} \log w(l) \;\leq\; |I| \cdot \log 2 - \sum_{p \in P} \log \frac{1}{|P|^2}$$
$$= O(|I| + |P| \log |P|) \;=\; O(n \log n). \quad (4.3)$$

Recalling that the actual time is the sum of the amortized time and the potential decrease, the total actual time is $O(\sum_{j=1}^{m} \log N(q_j) + m + n \log n)$. □

This theorem extends the static finger theorem for the traditional splay trees of [99] to a multidimensional setting. However, it is limited to the access of points that are in the given set. In a geometric setting, it is useful to extend this to points lying outside the set. The simplest generalization is to point location queries in the quadtree subdivision, that is, determining the leaf cell that contains a given query point. More generally, given a quadtree box Q, a *box query* returns the smallest outer box of any node of a splay quadtree that contains Q. (Point location queries arise as a special case corresponding to an infinitely small box.)

Our next result establishes a static finger theorem to box queries (and so applies to point location queries as well). Given a query box Q, starting from the root of the tree, we descend to the child node that contains Q until no such child exists. We then return the outer box of the current node, and if the current node is an internal node, we splay it, otherwise, we splay its parent node.

The following theorem establishes the static finger theorem for box queries.

Theorem 4.5.5 *Consider a set P of n points in $\mathbb{R}^d$ and a splay quadtree T for P.*

Let $Q_1, \ldots, Q_m$ *be a sequence of box queries. If* f *is any fixed point in* $\mathbb{R}^d$*, the total time to answer these queries is* $O\big(\sum_{j=1}^{m} \log N_f(Q_j) + m + n\log n\big)$*, where* $N_f(Q_j)$ *is the number of leaf cells overlapping the ball* $b(f, \text{dist}(f, Q_j))$ *in the subdivision induced by* T*.*

Proof: Since f is fixed, for each leaf cell z, let $N(z) = N_f(z)$. We assign a weight of $w(z) = 1/N(z)^2$ to each leaf cell z, where

$$N(z) = |\{z' \,:\, (z' \text{ is a leaf cell}) \;\wedge\; (\text{dist}(f, z') \leq \text{dist}(f, z))\}|,$$

that is, $N(z)$ is the number of leaf cells overlapping a ball of radius $\text{dist}(f, z)$ centered at f. We assign each leaf cell's weight to the weight of the corresponding leaf node, and we assign a weight of zero to each internal node.

For the analysis of the amortized time, we first compute the size $s(t)$ of the root t. This is the sum of the weights of all the leaf cells. As in Equation (4.2) of Theorem 4.5.4, the total weight of the leaf cells is $\sum_{z \in Z} w(z) < 2$. Let u_j be the internal node splayed by a query Q_j. The node u_j has a leaf cell, denoted by z_j satisfying $\text{dist}(f, z_j) = \text{dist}(f, cell(u_j))$. The size $s(u_j)$ of u_j is at least $w(z_j) = 1/N(z_j)^2$. Thus, $s(u_j) \geq w(z_j) = 1/N(z_j)^2$.

As in Equation (4.3) in Theorem 4.5.4, the net amortized time for the sequence of all m queries is

$$O\left(\sum_{j=1}^{m} (9(\log s(t) - \log s(u_j)) + 1\right) = O\left(\sum_{j=1}^{m} \log N(z_j) + m\right).$$

Note that Q_j is entirely included in $cell(u_j)$. This implies that

$$\mathrm{dist}(f, z_j) \;=\; \mathrm{dist}(f, cell(u_j)) \;\leq\; \mathrm{dist}(f, Q_j).$$

Thus, $N(z_j) \leq N(Q_j)$, and the net amortized time is $O(\sum_{j=1}^{m} \log N(Q_j) + m)$.

As in Equation (4.3) in Theorem 4.5.4, the decrease in potential over the m queries is $O(|I| + |Z| \log |Z|)$, which is $O(n \log n)$ since the size of the tree is linear in n. Thus, the actual total time is $O(\sum_{j=1}^{m} \log N(Q_j) + m + n \log n)$. □

Theorem 4.5.5 expresses the time for a series of box queries based on the number of leaf cells. In some applications, the induced subdivision is a merely a byproduct of a construction involving a set of P of points. In such case, it is more natural to express the time in terms of points, not quadtree cells. Given a set of box queries that are local to some given finger point f, we wish to express total access time as the function of the number of points of P in an appropriate neighborhood of f. As before, our analysis is based on quadtree leaf cells that are closer to f than the query box. The problem is that such leaf cells may generally contain points are very far from f, relative to the query box. To handle this, we allow a constant factor expansion in the definition of local neighborhood. Consider a ball containing the closest points of all given boxes from f (see Fig. 4.8(a)). Given a point set P, a finger point f, any constant c, and the box queries $Q_1, \ldots, Q_m$, the *working set* W_{box} with respect to P, f, c, and the box queries is defined:

$$W_{\text{box}}(P, f, c, Q_1, \ldots, Q_m) = \{p : p \in b(f, r_{\text{box}}(1+c)) \cap P,\ r_{\text{box}} = \max_j \mathrm{dist}(f, Q_j)\}.$$

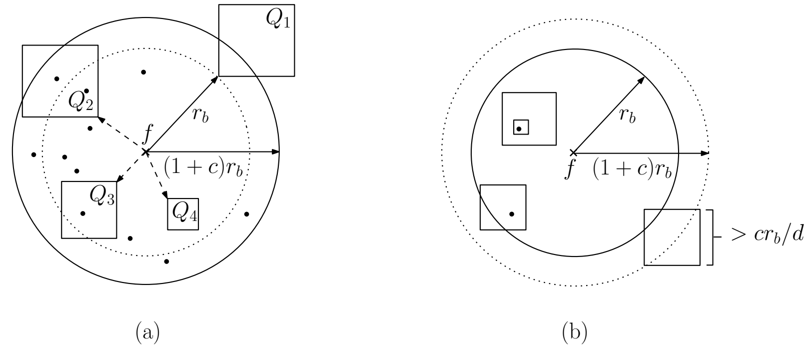


Fig. 4.8: (a) Working set, (b) the leaf cells overlapping a ball

Theorem 4.5.6 *Let* $Q_1, \ldots, Q_m$ *be the sequence of box queries. For any point* $f \in \mathbb{R}^d$ *and any positive constant* c*, the total time to answer these queries is* $O(m \log(|W_{\text{box}}| + (\frac{1}{c})^{d-1}) + n \log n)$.

Proof: By Theorem 4.5.5, the total time is $O(\sum_{j=1}^{m} \log N(Q_j) + m + n \log n)$. We will show that $N(Q_j)$ $(1 \leq j \leq m)$ is bounded by $O(|W_{\text{box}}| + (\frac{1}{c})^{d-1})$. Since $\text{dist}(f, Q_j) \leq r_{\text{box}}$, $N(Q_j)$ is at most the number of leaf cells overlapping a ball $b(f, r_{\text{box}})$. We distinguish between those overlapping leaf cells depending on whether the leaf cell is entirely included in the extended ball by $b(f, r_{\text{box}}(1+c))$, which we denote by b^+.

We first consider the number of leaf cells entirely contained within b^+ among leaf cells overlapping a ball $b(f, r_{\text{box}})$. Each leaf has a point or an inner box. The number of leaf cells with a point, entirely included in b^+ is at most the number of points in b^+, that is $|W_{\text{box}}|$. The number of leaf cells with an inner box, entirely contained within b^+ is also at most $|W_{\text{box}}|$ since each inner box has at least one point that causes it to be created. Thus, the number of leaf cells entirely contained within a ball b^+ is $O(|W_{\text{box}}|)$.

Next we consider the number of leaf cells that intersect the boundary of b^+ among leaf cells overlapping a ball $b(f, r_{\text{box}})$. Since these leaf cells have a distance from f less than r_{box}, and they intersect the boundary of a ball with radius $r_{\text{box}}(1+c)$, the size of each such cell is at least $r_{\text{box}}c/d$ (Fig. 4.8(b)). By the packing lemma of [11], the number of leaf cells, each of size at least $r_{\text{box}}c/d$, that intersect the boundary of a ball with radius $r_{\text{box}}(1+c)$ is $O((\frac{1}{c})^{d-1})$.

Therefore, the total number of leaf cells overlapping a ball $b(f, r_{\text{box}})$, that is, $N(Q_j)$ is $O(|W_{\text{box}}| + (\frac{1}{c})^{d-1})$ as desired. □

We will also analyze the time for answering approximate nearest neighbor and range searching queries. First, we consider approximate nearest neighbor queries using splay quadtrees. Let q be the query point, and let $\varepsilon > 0$ be the approximating factor. We apply a variant of the algorithm of [13]. We use a priority queue U and maintain the closest point p to q. Initially, we insert the root of the splay quadtree into U, and set p to a point infinitely far away. Then we repeatedly carry out the following process. First, we extract the node v with the highest priority from the queue, that is, the node closest to the query point. If the distance from v's cell to q exceeds $\text{dist}(q, p)/(1 + \varepsilon)$, we stop and return p. Since no subsequent point in any cell to be encountered can be closer to q than $\text{dist}(q, p)/(1+\varepsilon)$, p is an ε-approximate nearest neighbor. Otherwise, we descend v's subtree to visit the leaf node closest to the query point. As we descend the path to this leaf, for each node u that is visited, we compute the distance to the cell associated with u's siblings and then insert these siblings into U. If the visited leaf node contains a point, and the distance from q to

this point is closer than p, we update p to this point. Finally, we splay the parent of this leaf node.

Consider any sequence of m approximate nearest neighbor queries, $q_1, \ldots, q_m$. The working set for these queries can be defined to be

$$\begin{aligned} & W_{\text{ann}}(P, f, c, q_1, \ldots, q_m) \\ & \quad = \{p : p \in b(f, r_{\text{ann}}(1+c)) \cap P,\ r_{\text{ann}} = \max_j(\text{dist}(f, q_j) + \text{dist}(q_j, \text{NN}(q_j))\}. \end{aligned}$$

The following theorem shows that the time to answer these sequence of queries can be related to the size of this working set.

Theorem 4.5.7 *Let $q_1, \ldots, q_m$ be a sequence of ε-approximate nearest neighbor queries. For any point $f \in \mathbb{R}^d$ and any positive constant c, the total time to answer $q_1, \ldots, q_m$ is $O\big(m(\frac{1}{\varepsilon})^d \log\left(|W_{\text{ann}}| + \left(\frac{1}{c}\right)^{d-1}\right) + n \log n\big)$.*

Proof: As in the analysis of Theorem 4.5.5, we assign a weight of $w(z) = 1/N(z)^2$ to each leaf cell z, where $N(z)$ is the number of leaf cells overlapping a ball of radius $\text{dist}(f, z)$ centered at f, and we assign a weight of zero to each internal node.

Consider an approximate nearest neighbor query, q_j. Let $Q_1, \ldots, Q_k$ be the sequence of outer boxes of leafs visited by the algorithm in the order in which they are visited. The time to answer q_i is bounded to the time to answer box queries $Q_1, \ldots, Q_k$. By the analysis of Theorem 4.5.6, the amortized time for the jth query is $O(k \log(|W_{\text{box}}| + (\frac{1}{c})^{d-1}))$, where $W_{\text{box}} = \{p : p \in b(f, r_{\text{box}}(1+c)) \cap P,\ r_{\text{box}} = \max_i \text{dist}(f, Q_i)\}$. To complete the proof, we show that $W_{\text{box}} = W_{\text{ann}}$.

By the nature of priority search, the distances from the query point q_j to $Q_1, \ldots, Q_k$ increases monotonically. Thus, $r_{\text{box}} = \max_i \text{dist}(f, Q_i) = \text{dist}(f, Q_k)$. The distance from q_j to Q_k is at most

$$\frac{\text{dist}(q_j, \text{ANN}(q_j))}{(1+\varepsilon)} \leq \text{dist}(q_j, \text{NN}(q_j)) = r_{\text{ann}}.$$

If not, the algorithm terminates, and Q_k would not be visited. Thus, $r_{\text{box}} = r_{\text{ann}}$, that is, $W_{\text{box}} = W_{\text{ann}}$.

The number of leaf cells k visited by the algorithm is $O((\frac{1}{\varepsilon})^d)$ by the packing lemma of [13]. Thus, the amortized time for a query q_j is $O(k \log(|W_{\text{ann}}| + (\frac{1}{c})^{d-1})) = O((\frac{1}{\varepsilon})^d \log(|W_{\text{ann}}| + (\frac{1}{c})^{d-1}))$, and the net amortized time of all m queries is $O(m(\frac{1}{\varepsilon})^d \log(|W_{\text{ann}}| + (\frac{1}{c})^{d-1}))$.

By the same analysis as Theorem 4.5.5, the decrease in potential over the m queries is $O(n \log n)$. Thus, the total query time is $O(m(\frac{1}{\varepsilon})^d \log(|W_{\text{ann}}| + (\frac{1}{c})^{d-1}) + n \log n)$. □

Next, let us consider approximate range searching using the splay quadtree. For a given query range R and $\varepsilon > 0$, define an inner range R^- and an outer range R^+ to be the erosion and the dilation, respectively, of R by a distance of at least $\varepsilon \cdot \mathit{diam}(R)$. The approximate range searching returns the sum of weights of all points in a subset P' such that

$$P \cap R^- \subseteq P' \subseteq P \cap R^+.$$

As with the quadtreap [80], the tree maintains at each node v the total weight of the points in the associated subtree. Given a node v, let P_v denote the points of P lying within v's subtree, and let $\mathrm{wgt}(v)$ denote their total weight.

Initially, we insert the root of the splay quadtree into the queue, U. Then we repeatedly carry out the following procedure. First, we extract the node v from the queue. Then if the cell associated with this node intersects the query range R, we descend this subtree until we encounter a node that does not intersect R or a leaf node. Such a node is called a *terminal* node. When we find a terminal node, we first splay its parent, and then determine whether its weight is to be included in the output. Recall that each leaf cell contains a point or an inner box. If the terminal node is a leaf and has a point, we check whether the point is contained in R. If so, we return its weight. Otherwise, the node does not intersect R, that is, its cell is either entirely inside R^+ or outside R^-. If the cell is inside R^+, we return the sum of weights of all points in its subtree. As we descend the path to the terminal node, for each node v that is visited, we insert its sibling nodes into the queue. The procedure is shown in Algorithm 8.

Consider a sequence of m approximate range searching queries, $R_1, \ldots, R_m$. In order to define a working set, consider a ball containing all the query ranges. Let r_{AR} be the radius of the smallest ball. The working set, W_{rng} for approximate range

Algorithm 8 Range Query

Input: A range R, given with an inner range R^- and an outer range R^+, and a node v in the splay quadtree

Output: The total weight of points in the approximate range

1: **function** Query(R, v)
2: $U \leftarrow \{\mathrm{root}(T)\}$
3: **while** $U \neq \emptyset$ **do**
4: $v \leftarrow \mathit{dequeue}(U)$
5: **while** v is an internal node and cell(v) intersects R **do**
6: $v \leftarrow \mathrm{left}(v)$
7: $\mathit{enqueue}(U, \mathrm{right}(v))$
8: $\mathit{enqueue}(U, \mathrm{outer}(v))$
9: $\mathit{splay}(p(v))$
10: **if** v is a leaf and has a point **then**
11: **if** $\mathit{point}(v) \in R^+$ **then return** wgt($\mathit{point}(v)$)
12: **else if** cell(v) $\subseteq R^+$ **then return** wgt(v)

queries can be defined to be

$$W_{\mathrm{rng}}(P, f, c, R_1, \ldots, R_m)$$
$$= \{p : p \in b(f, r_{\mathrm{rng}}(1+c)) \cap P,\ r_{\mathrm{rng}} = \max_j \max_{q \in R_j} \mathrm{dist}(f, q)\}.$$

The following theorem shows that the time to answer all the queries of the sequence can be related to the size of this working set.

Theorem 4.5.8 *Let $R_1, \ldots, R_m$ be the sequence of query ranges for ε-approximate range searching. For any point f in $\mathbb{R}^d$ and any positive constant c, the total time to answer $R_1, \ldots, R_m$ is $O\big(m\left(\frac{1}{\varepsilon}\right)^{d-1} \log(|W_{\mathrm{rng}}| + \left(\frac{1}{c}\right)^{d-1}) + n \log n\big)$.*

Proof: As in previous analyses, we will assign a weight to each node of the splaying quadtree. (This is not to be confused with the weight is used in answering the queries.) As in the analysis of Theorem 4.5.5, we assign a weight of $w(z) = 1/N(z)^2$

to each leaf cell z, where $N(z)$ is the number of leaf cells overlapping a ball of radius $\mathrm{dist}(f, z)$ centered at f, and we assign a weight of zero to each internal node.

The amortized time for R_j is bounded by the time to splay all the parents of all its terminal nodes. Let $u_1, \ldots, u_k$ be the sequence of parents of the terminal nodes, and let Q_i be the outer box of u_i ($1 \leq i \leq k$). The time to answer R_j is bounded to the time to answer box queries $Q_1, \ldots, Q_k$. By the analysis of Theorem 4.5.6, the amortized time for R_j is $O(k \log(|W_{\mathrm{box}}| + (\frac{1}{c})^{d-1}))$, where $W_{\mathrm{box}} = \{p : p \in b(f, r_{\mathrm{box}}(1+c)) \cap P,\ r_{\mathrm{box}} = \max_i \mathrm{dist}(f, Q_i)\}$. Here, $r_{\mathrm{box}} = \max_i \mathrm{dist}(f, Q_i) \leq \max_{q \in R_j} \mathrm{dist}(f, q) \leq r_{\mathrm{rng}}$, since clearly the parent node u_i of any terminal node intersects the query range. Thus, $W_{\mathrm{box}} = W_{\mathrm{rng}}$.

The number of terminal nodes k visited by the algorithm is $O((\frac{1}{\varepsilon})^{d-1})$ by the packing lemma of [11]. Thus, the amortized time for the jth query is

$$O\left(k \log\left(|W_{\mathrm{rng}}| + \left(\frac{1}{c}\right)^{d-1}\right)\right) = O\left(\left(\frac{1}{\varepsilon}\right)^{d-1} \log\left(|W_{\mathrm{rng}}| + \left(\frac{1}{c}\right)^{d-1}\right)\right),$$

and the net amortized time of all m queries is $O(m(\frac{1}{\varepsilon})^{d-1} \log(|W_{\mathrm{rng}}| + (\frac{1}{c})^{d-1}))$. The decrease in potential over the m queries is $O(n \log n)$ by the same analysis as in Theorem 4.5.5. Thus, the actual total time is $O(m(\frac{1}{\varepsilon})^{d-1} \log(|W_{\mathrm{rng}}| + (\frac{1}{c})^{d-1}) + n \log n)$. □

Chapter 5

Output-Sensitive Well-Separated Pair Decompositions for Dynamic Point Sets[1]

5.1 Introduction

The well-separated pair decomposition (WSPD) is a fundamental structure in computational geometry. Given a set P of n points in d-dimensional space and a positive separation parameter s, an s-WSPD is a concise representation of all the $O(n^2)$ pairs of P requiring only $O(s^d n)$ storage. Worst-case arguments suggest that the addition or deletion of a single point could result in the generation (or removal) up to $\Omega(s^d)$ pairs, which can be unacceptably high in many applications. The actual number of well separated pairs can be significantly smaller in practice, particularly when the points are well clustered. This suggests the importance of being able to respond to insertions and deletions in a manner that is output sensitive, that is, whose running time depends on the actual number of pairs that have been added or removed.

In this chapter, we present output-sensitive algorithms for maintaining the WSPD for a dynamic point set under insertion and deletion. The following theorem presents our main result. We assume that point coordinates are represented as fixed-point binary numbers that support bit-wise operations (such as boolean-and,

[1]The material appearing in the chapter is based on the paper "Output-Sensitive Well-Separated Pair Decompositions for Dynamic Point Sets" [88].

boolean-or, and shift) in $O(1)$ time. Our deletion operation employs a form of "lazy updating," where some housekeeping operations are deferred until later, which results in amortized running time bounds.

Theorem 5.1.1 *Given a dynamic set $P \subset \mathbb{R}^d$ and a parameter $s \geq 1$, it is possible to maintain an s-WSPD for P in the L_∞ metric that supports the following operations and running times:*

- Insert point: $O(\log n + m)$ *(worst case)*
- Delete point: $O(\log n + m)$ *(amortized)*

where m denotes the number of newly created (resp., deleted) pairs in the case of insertion (resp., deletion). (The hidden constants depend on d, but are independent of s.)

The remainder of the chapter is organized as follows. In Section 5.2, we present basic definitions and introduce our stronger notion of separation. In Section 5.3 we present static algorithms and utilities for constructing the WSPD. These are used by our update algorithms, which are given in Section 5.4.

5.2 Preliminaries

We begin by recalling some basic concepts related to quadtrees, upon which our constructions are based. Let $\mathbb{U}^d = [0, 1)^d$ denote the half-open d-dimensional unit hypercube. We define a *quadtree box* by the following recursive process. $\mathbb{U}^d$ is a quadtree box, and given any quadtree box b, each of the 2^d regions that result by

subdividing b into congruent (half-open) hypercubes is also a quadtree box. These 2^d boxes are called the *children* of b. The subdivision process naturally defines a 2^d-ary partition tree whose nodes are correspond to quadtree boxes. Henceforth, we use the term "box" to mean "quadtree box."

Consider an n-element point set $P \subset \mathbb{U}^d$. Such a set naturally induces a quadtree subdivision, by repeatedly subdividing each box that contains two or more points of P. Because the size of the resulting subdivision may be much larger than n, it is common to compress *trivial paths*, in which each internal node has only one nonempty child. A *compressed quadtree* is obtained by (i) removing all leaf boxes that contain no point of P and then (ii) replacing any maximal trivial path with a single edge that goes to the first descendent with two or more children. In any fixed dimension, it is well known that such a quadtree is of size $O(n)$ and can be constructed in time $O(n \log n)$ (see, e.g. [30, 63]). Given a node u in a compressed quadtree, let $\text{par}(u)$ denote its parent.

Each node of a compressed quadtree is naturally associated with a box, which is often called its cell. Following [63], we define the *level* of a box u, denoted $\text{lev}(u)$, to be the base-2 logarithm of its side length, and we define the level of a node to be the level of its associated box. (Throughout, we use lg to denote the base-2 logarithm. Because we start with $\mathbb{U}^d$, a node's level is the negation of its depth in a standard quadtree.) It is convenient to think of a point as a degenerate box of side length zero, that is, a box at level $-\infty$. (We will often blur the distinction between a node and its associated box by using the same name for both.) Given an integer $k \geq 0$ and a box u such that $\text{lev}(u) \leq -k$, let $u^{\uparrow k}$ denote the (unique) box at level

lev(u) + k that contains u.

Given a compressed quadtree for P and a box b, define node(b) to be node of the highest level in the tree whose associated box is contained within (or is equal to) b. If no such node exists (which happens if b contains no point of P) then we assume that node(b) returns a special *null* value. By storing the quadtree in a balanced structure, e.g., as a topology tree [50] or a link-cut tree [98], node(b) can be computed in $O(\log n)$ time. This is called a *box query*.

As mentioned in the introduction, we compute the WSPD assuming the L_∞ metric. Given two boxes u and v, define dist(u, v) to be the minimum L_∞ distance between any two points, one from u and one from v. Since the radius of the smallest L_∞ ball enclosing a box is half the box's side length, it follows that u and v are s-well separated if $\text{dist}(u, v) \geq s \cdot 2^\ell/2$, where $\ell = \max(\text{lev}(u), \text{lev}(v))$.

5.2.1 Strong Separation

For the sake of simplicity and efficiency, we will need a stronger notion of separation in our algorithms. (We shall see in Lemma 5.2.3 below that this alters the size of the WSPD by only a constant factor.) We start by introducing a quadtree-based notion of separation. We say that two boxes u and v are *homogeneously s-well separated* if there exist two (quadtree) boxes u' and v' of equal level such that $u \subseteq u'$ and $v \subseteq v'$, and u' and v' are s-well separated. If two boxes u and v are homogeneously s-well separated, then clearly they are s-well separated. (As shown in Fig. 5.1(a), the converse does not hold.)

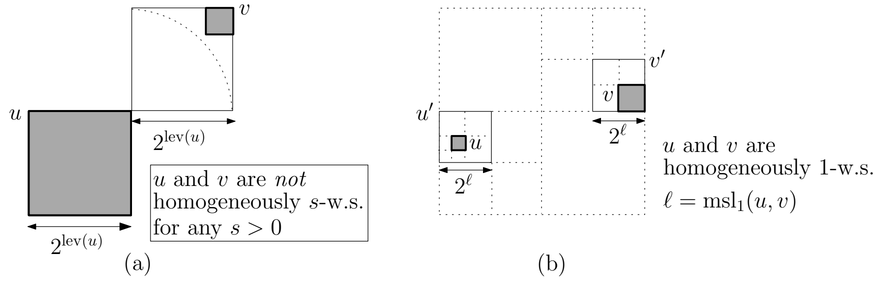


Fig. 5.1: Homogeneous separation.

To minimize the size of the WSPD, it is important that the well-separated pairs are maximal in some sense. If u and v are homogeneously s-well separated, there exists a unique maximum level ℓ at which u' and v' reside. We call this the *maximum separation level* and denote it by $\text{msl}_s(u, v)$ (see Fig. 5.1(b)).

Assuming a model of computation supporting bit manipulations, $\text{msl}_s(u, v)$ can be computed in constant time, for any boxes u and v. (For our algorithms, we will only need $\text{msl}_1(u, v)$.)

Lemma 5.2.1 *Assume we have a model of computation that supports in* $O(1)$ *time the bitwise operations boolean-or ("$\vee$"), complement ("$\overline{x}$"), and exclusive-or ("$\oplus$"), left shift ("$\ll$"), and integer base-2 logarithm (that is, the index of the most significant bit) on point coordinates. Given two boxes u and v that are homogeneously 1-well separated, it is possible to compute* $\text{msl}_1(u, v)$ *in* $O(1)$ *time.*

Proof: Consider a homogeneous 1-well separated pair $\{u, v\}$. To motivate our construction, let $\ell = \text{msl}_1(u, v)$, and let u' and v' be the boxes of level ℓ containing u and v, respectively. Let $p' = (p'_1, \ldots, p'_d)$ and $q' = (q'_1, \ldots, q'_d)$ denote the lower left corners of u' and v', respectively. Since u' and v' are 1-well separated, we have $\text{dist}(u', v') \geq 2^\ell/2$, and by basic properties of boxes and the definition of the L_∞

distance, we have $\max_i |p'_i - q'_i| > 2^\ell$. Therefore, $\max_i \left|\frac{p'_i}{2^\ell} - \frac{q'_i}{2^\ell}\right| > 1$. Note that for any point p in u', $\forall i$, $\left\lfloor \frac{p_i}{2^\ell} \right\rfloor = \frac{p'_i}{2^\ell}$. Thus, letting p and q be the lower left corners of u and v, respectively, we have $\forall i$, $\left\lfloor \frac{p_i}{2^\ell} \right\rfloor = \frac{p'_i}{2^\ell}$, and $\left\lfloor \frac{q_i}{2^\ell} \right\rfloor = \frac{q'_i}{2^\ell}$. Therefore, to compute $\ell = \mathrm{msl}_1(u, v)$ it suffices to compute the largest $\ell \leq 0$ such that

$$\max_{1 \leq i \leq d} \left| \left\lfloor \frac{p_i}{2^\ell} \right\rfloor - \left\lfloor \frac{q_i}{2^\ell} \right\rfloor \right| \; > \; 1. \tag{5.1}$$

Algorithm 9 Find $\mathrm{msl}_1(u, v)$

Input: Two boxes u and v that are homogeneously 1-well separated
Output: The maximum separation level of u and v, that is, $\mathrm{msl}_1(u, v)$

1: **function** findMaxLevel(u, v)
2: $\quad p$: the lower left corner of u
3: $\quad q$: the lower left corner of v
4: $\quad m \leftarrow \arg\max_i |p_i - q_i|$
5: $\quad x \leftarrow \max(p_m, q_m) \qquad y \leftarrow \min(p_m, q_m)$
6: $\quad j_1 \leftarrow \lfloor \lg(x \oplus y) \rfloor$
7: $\quad x \leftarrow x \ll j_1 \qquad y \leftarrow y \ll j_1$
8: $\quad j_2 \leftarrow \lfloor \lg(x \vee \bar{y}) \rfloor$
9: $\quad$ **return** $j_1 + j_2$

The code implementing this is presented in Algorithm 9. Let p and q denote the lower left corners of u and v, respectively. The procedure first computes the largest absolute coordinate m of $p - q$ ($1 \leq m \leq d$). We let x and y be the maximum and minimum of p_m and q_m, respectively. Recalling that the points lie within the half-open unit hypercube $[0, 1)^d$, $\left\lfloor \frac{x}{2^j} \right\rfloor$ yields the first j bits in x's bit representation. We compare the leading j bits of x and y for increasing values of j. Let j_1 be the first position where x and y have different bit values (see Fig. 5.2). Since x is greater than y, the j_1th bits of x and y are 1 and 0, respectively. Line 6

finds this position j_1 by computing the index of the most significant bit of $x \oplus y$. We then shift these matching bits off the left end of the bit string. (We assume that bits shifted to the left of the decimal point are discarded.)

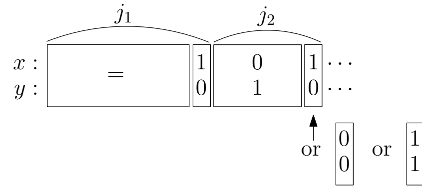


Fig. 5.2: Find the maximum separation level of homogeneous 1-well separated pair

Next, we find the first position, j_2 following this where x's bit is not 0 and y's bit is not 1 (see Fig. 5.2). This is done by computing the index of the most significant bit of $x \vee \bar{y}$ (Line 8). The result corresponds to the first position where their difference is greater than 1. It follows that $j_1 + j_2$ is the desired level. $\square$

In order to define our stronger notion of separation, let $s \geq 1$ denote the WSPD separation factor, which will be fixed henceforth, and define

$$\sigma \;=\; \max\left(\left\lceil \lg \frac{s}{2} \right\rceil, 0\right).$$

We say that two boxes u and v are *strongly s-well separated* if $u^{\uparrow\sigma}$ and $v^{\uparrow\sigma}$ are homogeneously 1-well separated. As in homogeneous well separation, there exists a unique maximum separation level ℓ at which two such boxes are strongly s-well separated, which we denote by $\mathrm{msl}_s^*(u, v)$. Clearly, $\mathrm{msl}_s^*(u, v) = \mathrm{msl}_1(u^{\uparrow\sigma}, v^{\uparrow\sigma}) - \sigma$. Our next lemma shows that this is indeed a stronger notion of separation.

Lemma 5.2.2 *If two boxes u and v are strongly s-well separated, then they are*

s-well separated.

Proof: If u and v are strongly s-well separated, then by definition, $u^{\uparrow\sigma}$ and $v^{\uparrow\sigma}$ are homogeneously 1-well separated. Note that if any two boxes of equal level are not adjacent, they are separated from each other by an (L_∞) distance of at least their side length. Letting $\ell = \max(\text{lev}(u), \text{lev}(v))$, we have

$$\begin{aligned} \text{dist}\big(u^{\uparrow\sigma}, v^{\uparrow\sigma}\big) &\geq 2^{\max(\text{lev}(u^{\uparrow\sigma}),\text{lev}(v^{\uparrow\sigma}))} = 2^{\max(\text{lev}(u)+\sigma,\text{lev}(v)+\sigma)} \\ &= 2^{\sigma} \cdot 2^{\ell} \geq \frac{s}{2} \cdot 2^{\ell}. \end{aligned}$$

Note that since $u^{\uparrow\sigma}$ and $v^{\uparrow\sigma}$ contain u and v, respectively, we have $\text{dist}(u, v) \geq \text{dist}\big(u^{\uparrow\sigma}, v^{\uparrow\sigma}\big)$. Therefore, $\text{dist}(u, v) \geq s \cdot 2^{\ell}/2$, which implies that u and v are s-well separated. □

The notion of maximality can be generalized to this context. Consider two nodes u and v that are strongly s-well separated. Let $u' = \text{node}(u^{\uparrow\sigma})$ and $v' = \text{node}(v^{\uparrow\sigma})$. By definition, u' and v' are homogeneously 1-well separated. We say that the pair (u, v) is *maximal* if $\text{par}(u')$ and $\text{par}(v')$ are not homogeneously 1-well separated. Our algorithms generate only maximally well-separated pairs.

If we modify the definition of s-WSPD by replacing the standard notion of separatedness with this stronger notion, we obtain a structure called a *strong s-well separated pair decomposition* (strong s-WSPD). The following lemma shows that by using this stronger notion of separation, we sacrifice only a constant factor in the size of the WSPD, in comparison to *any* standard s-WSPD and in *any* Minkowski

metric.

Lemma 5.2.3 *Given a points set P in $\mathbb{R}^d$ and $s \geq 1$, let Ψ be a strong s-WSPD (in the L_∞ metric) consisting of maximal pairs. Let Ψ' be any (standard) s-WSPD in the L_p metric for any $p \geq 1$. Then, there exists a constant c (depending on d but independent of s and p) such that $|\Psi| \leq c \cdot |\Psi'|$.*

Proof: Because Ψ consists only of maximal pairs, it suffices to show that each well-separated pair $\{A_i, B_i\} \in \Psi'$ can be covered by at most a constant number of strong s-well-separated pairs. By definition, A_i and B_i can each be enclosed within an L_p ball of radius r such that the closest L_p distance between these balls is at least sr. In $\mathbb{R}^d$, if the L_p distance between two points is z, then their L_∞ distance is at least $z/d^{1/p} \geq z/d$ (since $p \geq 1$). Therefore, the closest L_∞ distance between these balls is at least sr/d.

Let ℓ be the largest integer (possibly negative) such that $2^\ell \leq sr/(3d)$. Consider the grid of quadtree boxes of side length 2^ℓ. The L_∞ diameter of any box of the grid is 2^ℓ. Let u' and v' be any boxes of this grid that have a nonempty intersection with A_i and B_i, respectively. By the triangle inequality, the closest L_∞ distance between u' and v' is at least

$$\frac{sr}{d} - \mathrm{diam}(u') - \mathrm{diam}(v') \;=\; \frac{sr}{d} - 2 \cdot 2^\ell \;\geq\; \frac{sr}{d} - 2\frac{sr}{3d} \;=\; \frac{sr}{3d} \;\geq\; 2^\ell.$$

It follows that u' and v' are homogeneously 1-well separated.

Let u and v be any quadtree boxes at level $\ell - \sigma$ that have a nonempty intersection with A_i and B_i, respectively. Clearly, $u^{\uparrow\sigma}$ and $v^{\uparrow\sigma}$ intersect A_i and B_i,

respectively, and so by the above analysis, they are homogeneously 1-well separated. Therefore u and v are strongly s-well separated. The side length of these boxes is

$$2^{\ell-\sigma} \geq \frac{2^{\ell}}{2^{1+\lg\frac{s}{2}}} = \frac{2^{\ell}}{s}.$$

By definition of ℓ, we have $2^{\ell} > sr/(6d)$, and therefore u and v are both of side length at least $r/(6d)$. It follows from a standard packing argument, that the number of such boxes that can have a nonempty intersection with an L_p ball of radius r is a constant γ. (In general, γ depends on the dimension but not on s or p. A more careful analysis reveals that $\gamma \leq \lceil 1 + 6d \rceil^d$ [12].)

Since at most γ quadtree boxes at level $\ell - \sigma$ suffice to cover each of A_i and B_i, it follows that the number of strong s-well-separated pairs that cover the pairs $A_i \times B_i$ is at most γ^2. In summary, each (standard) s-well-separated pair of Ψ' can be covered by at most $\gamma^2 = O(1)$ strong s-well-separated pairs of Ψ, which completes the proof. □

Throughout the remainder of the paper, we use the term "s-well separated" to mean strongly s-well separated and we use s-WSPD to mean strong s-WSPD.

5.3 Construction and Utilities

In this section, we present the basic construction algorithm for computing an s-WSPD for any $s \geq 1$. Our construction algorithm employs the standard recursive process for computing WSPDs (see Algorithm 10). Given a pair of nodes u and v,

it first dispenses with the trivial cases (either node is empty or the two nodes are the same leaf). If the nodes are homogeneously 1-well separated, then it computes $\ell = \text{msl}_1(u, v)$ and invokes a utility function, called findDescendants, that computes all the maximal descendants of u and v at level $\ell - \sigma$, where σ is the value used in the definition of strong separation (Section 5.2.1). It returns the cross product of the resulting nodes. Otherwise (if u and v are not homogeneously well-separated), it subdivides the node that is at the higher level of the tree and applies the procedure recursively to the children of this node.

Each well-separated pair is represented in the form $(\{x, y\}, \ell)$, where x and y are the nodes defining the well-separated pair, and $\ell = \text{msl}^*_s(x, y)$ is their maximum separation level. The initial call is findWSPD(r, r, s), where r is the root of the compressed quadtree, and s is the separation factor.

Algorithm 10 Construction of WSPD

Input: Two nodes u and v and a separation parameter $s \geq 1$
Output: All s-well separated pairs between the subtrees of u and v

1: **function** findWSPD(u, v, s)
2:     **if** $u = v$ and $\text{lev}(u) = -\infty$ **then return** $\emptyset$
3:     **if** u and v are homogeneously 1-well separated **then**
4:         $\ell \leftarrow \text{msl}_1(u, v)$ ▷ See findMaxLevel in Algorithm 9
5:         $\sigma \leftarrow \max\left(\left\lceil \lg \frac{s}{2} \right\rceil, 0\right)$
6:         $X \leftarrow$ findDescendants(u, ℓ)
7:         $Y \leftarrow$ findDescendants(v, ℓ)
8:         **return** $\bigcup_{x \in X, y \in Y} (\{x, y\}, \ell)$
9:     **if** $\text{lev}(u) \geq \text{lev}(v)$ **then return** $\bigcup_i$ findWSPD(u_i, v, s)
10:     **else return** $\bigcup_i$ findWSPD(u, v_j, s)

11: **function** findDescendants(u, ℓ)
12:     **if** $\text{lev}(u) \leq \ell$ **then return** $\{u\}$
13:     **return** $\bigcup_i$ findDescendants(u_i, ℓ)

We test whether two nodes u and v are homogeneously 1-well separated as follows. Let $\ell' = \max(\text{lev}(u), \text{lev}(v))$. Let u' and v' denote the respective containing boxes at this level. Then u and v are 1-well separated if and only if u' and v' are not adjacent. This can be determined in $O(1)$ time, using the same bit manipulations given in Lemma 5.2.1. Recall that each leaf is associated with a point, which we treat as a box at level $-\infty$. Therefore, two distinct leaves are always homogeneously 1-well separated.

We observe the following properties of the well-separated pairs generated by Algorithm 10. Recall that $\text{par}(u)$ denotes the parent of node u in the compressed quadtree.

Lemma 5.3.1 *Consider any nodes r_1 and r_2 of the compressed quadtree (possibly $r_1 = r_2$). Consider any pair $\{x, y\}$ generated by the call findWSPD(r_1, r_2, s). Let $\{u, v\}$ be the pair of homogeneously 1-well separated nodes that resulted in the generation of the pair (x, y) at Line 8 of Algorithm 10. Then*

(*i*) $\text{lev}(u) \leq \text{msl}_1(u, v)$, *and if* $u \neq r_1$, *then* $\text{msl}_1(u, v) < \text{lev}(\text{par}(u))$

(*ii*) $\text{lev}(x) \leq \text{msl}^*_s(x, y)$, *and if* $x \neq r_1$, *then* $\text{msl}^*_s(x, y) < \text{lev}(\text{par}(x))$

(Symmetrical bounds hold for v and y.)

Proof: First, consider (*i*). By the definition of homogeneous well separation, there exist u' and v' on level $\text{msl}_1(u, v)$ such that $u \subseteq u'$ and $v \subseteq v'$, and u' and v' are homogeneously 1-well separated. Therefore, $\text{lev}(u)$ is not greater than $\text{msl}_1(u, v)$, which establishes the first inequality. To prove the second inequality, suppose to

the contrary that $u \neq r_1$ and $\mathrm{msl}_1(u,v) \geq \mathrm{lev}(\mathrm{par}(u))$. This would imply that $\mathrm{par}(u) \subseteq u'$, and so $\mathrm{par}(u)$ and v would be homogeneously 1-well separated. By the nature of findWSPD, $\mathrm{par}(u)$ would never have been recursively subdivided, which yields the desired contradiction.

To prove (ii) observe that, by the definition of findDescendants, x and y are maximal nodes on a level not more than $\mathrm{msl}_s^*(x,y)$. Therefore, $\mathrm{lev}(x) \leq \mathrm{msl}_s^*(x,y) < \mathrm{lev}(\mathrm{par}(x))$. The analogous bounds for v and y hold by symmetry. □

The following lemma establishes the correctness of the algorithm.

Lemma 5.3.2 *Consider a point set P, a separation factor $s \geq 1$, and a compressed quadtree T storing P. Letting r denote T's root, the call findWSPD(r,r,s) returns an s-WSPD for P. In particular, for any pair $\{x,y\}$ returned, resulting x and y are maximally s-well separated.*

Proof: Consider any pair $\{x,y\}$ generated by the algorithm, and let $\{u,v\}$ be the pair of homogeneously 1-well separated nodes that resulted in the generation of this pair on Line 8. We have $\mathrm{msl}_s^*(x,y) = \mathrm{msl}_1(u,v) - \sigma$.

We first show that two boxes, $x^{\uparrow\sigma}$ and $y^{\uparrow\sigma}$ are homogeneously 1-well separated, and in particular, $\mathrm{msl}_1(x^{\uparrow\sigma}, y^{\uparrow\sigma}) = \mathrm{msl}_1(u,v)$. Let u' and v' be the boxes at level $\mathrm{msl}_1(u,v)$ containing u and v, respectively. Note that $\mathrm{lev}(x) \leq \mathrm{msl}_s^*(x,y)$ and $\mathrm{lev}(y) \leq \mathrm{msl}_s^*(x,y)$, and $x \subseteq u \subseteq u'$ and $y \subseteq v \subseteq v'$. Therefore

$$\begin{aligned}
\mathrm{lev}\left(x^{\uparrow\sigma}\right) &= \mathrm{lev}(x) + \sigma \leq \mathrm{msl}_s^*(x,y) + \sigma = \mathrm{msl}_1(u,v) = \mathrm{lev}(u') \\
\mathrm{lev}\left(y^{\uparrow\sigma}\right) &= \mathrm{lev}(y) + \sigma \leq \mathrm{msl}_s^*(x,y) + \sigma = \mathrm{msl}_1(u,v) = \mathrm{lev}(v').
\end{aligned}$$

Clearly, $x^{\uparrow\sigma}$ and $y^{\uparrow\sigma}$ are contained within u' and v', respectively. Since $x^{\uparrow\sigma} \subseteq u'$, $y^{\uparrow\sigma} \subseteq v'$, and u' and v' are homogeneously 1-well separated, it follows that $x^{\uparrow\sigma}$ and $y^{\uparrow\sigma}$ are homogeneously 1-well separated. The maximum separation level of $x^{\uparrow\sigma}$ and $y^{\uparrow\sigma}$ is $\mathrm{msl}_1(u, v)$.

Next, we show that $\{x, y\}$ is maximal. By Lemma 5.3.1(ii), $\mathrm{par}(x)^{\uparrow\sigma}$ is on a level higher than $\mathrm{msl}^*_s(x, y)+\sigma = \mathrm{msl}_1(u, v)$, and therefore it is not homogeneously 1-well separated with any node containing y. The same applies to $\mathrm{par}(y)^{\uparrow\sigma}$. Therefore the pair $\{x, y\}$ is maximal, which completes the proof. □

The following lemma is implied by Lemmas 5.2.3 and 5.3.2.

Lemma 5.3.3 *Given a separation factor $s \geq 1$, the number of pairs generated by findWSPD is $O(s^d n)$.*

The following lemma shows that, after constructing the compressed quadtree, the algorithm's running time is linear in the number of pairs generated.

Lemma 5.3.4 *Given an n-element point set P in $\mathbb{R}^d$ stored in a compressed quadtree and $s \geq 1$, findWSPD computes an s-WSPD in time linear in the number of pairs.*

Proof: Consider the computation tree whose nodes correspond the calls findWSPD. We say that a call is *terminal* if it makes no recursive calls to findWSPD, and we say that a terminal call is *effective* if it returns on Line 8. We will show that the total number of nodes in the resulting computation tree can be bounded by the number of effective terminal calls. To do this, we charge each noneffective terminal to the nonterminal call that generated it, that is, its parent in the computation

tree. Recalling that each internal node in the compressed quadtree has at least two children, it follows that each nonterminal call generates at least two calls, neither of which is a noneffective terminal. In other words, if we prune all noneffective terminals from the computation tree, each internal node has at least two children. Thus, the total number of nodes in the computation tree is linear in the number of effective terminals.

Now consider the running time of effective terminal call. For a 1-well separated pair, $\{u, v\}$ on Line 3, it finds all descendants of u and v of level less than or equal to $\mathrm{msl}_1(u, v) - \sigma$, where $\sigma = \max(\lceil \lg \frac{s}{2} \rceil, 0)$, by descending the tree (using recursive calls to findDescendants). It generates s-well separated pairs consisting of the cross product of these two sets of descendants. Clearly, the total running time of all the recursive calls to findDescendants is bounded by the total number of pairs it outputs. Thus, each effective terminal call runs in time linear in the resulting number of s-well-separated pairs. Therefore, the total running time to construct the WSPD is proportional to the final output size. □

5.4 Updates

In this section, we present the algorithms for inserting and deleting points into the WSPD. We first consider insertion in Section 5.4.1, deletion in Section 5.4.2, and we discuss maintenance of an internal data structure in Section 5.4.3.

5.4.1 Insertion

Consider the insertion of a new point p into the compressed quadtree. We first locate the node whose box contains the newly inserted point. By storing the compressed quadtree using an auxiliary structure, such as a topology tree [50], this can be done in $O(\log n)$ time. As observed by Fischer and Har-Peled [48], there are two cases depending on the location where the new point is inserted.

Case 1: (no new internal node) The new point p is stored in a leaf, denoted by x, that is hung directly beneath its parent (see Fig. 5.3(a)).

Case 2: (between the cells of the compressed node w and its parent) We create new leaf, x containing p, and a new node z, which has w and x among its children (see Fig. 5.3(b)).

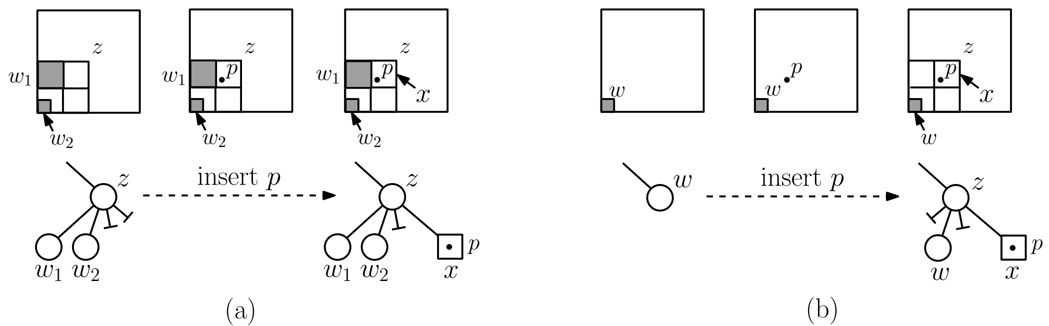


Fig. 5.3: Inserting a point into the compressed quadtree.

After the insertion of p, we need to update the well-separated pairs. Our objective is that updated pairs will be the same as would result by running the static construction algorithm findWSPD on the updated point set. The insertion procedure is given in Algorithm 11.

First, let us consider Case 2 (Fig. 5.3(b)) where the insertion resulted in the creation of node z. Note that z's cell contains the cell of x's sibling node w, which

existed prior to the insertion. Recall that each well-separated pair is represented by a triple $(\{x, y\}, \ell)$, where x and y are the two nodes defining the pair, and ℓ is the maximum separation level. Consider any well-separated pair $(\{w, v\}, \ell)$ that existed prior to the insertion, where $\ell \geq \text{lev}(z)$. If findWSPD were to be run on the quadtree containing p, then because z will be encountered before w, and so it follows that (after insertion) this pair involves z rather than w. We replace each such pair with the new pair $(\{z, v\}, \ell)$. Note that since $\ell \geq \text{lev}(z)$, and ℓ is not affected by the insertion, ℓ is not changed.

Next, consider Case 1 (Fig. 5.3(a)), where x denotes the leaf containing the new point. Consider any s-well separated pair $\{x, y\}$ generated by findWSPD on the updated point set. Since z and y are not strongly s-well separated (since otherwise the algorithm would have generated the pair $\{z, y\}$), we have $z^{\uparrow\sigma}$ and $y^{\uparrow\sigma}$ are not homogeneously 1-well separated. Let B denote the set of boxes at level $\text{lev}\big(z^{\uparrow\sigma}\big)$ that are not homogeneously 1-well separated with respect to $z^{\uparrow\sigma}$. Clearly, $y^{\uparrow\sigma}$ is contained within some box of B.

To compute these new pairs, for each $b \in B$, we find the corresponding node in the compressed quadtree by invoking $\text{node}(b)$. Recall that this returns the largest node in the compressed tree that is contained within b. (If the result is null, we simply ignore the result.) We determine the well-separated pairs involving the new leaf as follows. For each $b \in B$, we invoke $\text{findWSPD}(x, \text{node}(b), s)$. We take the union of all the resulting pairs and add them to the WSPD.

The following lemma establishes the correctness this procedure. (We will present the running time later.)

Algorithm 11 Insertion of a point p for output sensitive WSPD

Input: A point, p and a separation factor, $s \geq 1$

Output: All s-well separated pairs related to new leaf including p

```
 1: function insert(p, s)
 2:     Insert p
 3:     x : a new leaf including p
 4:     z : parent of x
 5:     if z is new then
 6:         w : x's sibling
 7:         ∀ s-WSPs ({w, v}, ℓ) with ℓ ≥ lev(z)), change ({w, v}, ℓ) to ({z, v}, ℓ).
 8:     σ ← max(⌈lg s/2⌉ , 0)
 9:     B : set of boxes adjacent to z^{↑σ} including z^{↑σ}
10:     return ⋃_{b∈B} findWSPD(x, node(b), s)
```

Lemma 5.4.1 *Function insert of Algorithm 11 correctly updates the s-WSPD.*

Proof: Let T denote the compressed quadtree before p's insertion, and let T' denote the compressed quadtree after p's insertion. Let W be the set of well-separated pairs generated by findWSPD when run on T, and W' be the analogous set on T'. We will show that Algorithm 11 converts W into W'. We consider the same two cases mentioned above, and we use x, z, and w as defined above. There are only two types of pairs that would be affected by the insertion, those involving x itself and those involving the newly added internal node z.

First, let us consider the pairs involving z (Case 2). Consider any well-separated pair, $(\{z, y\}, \ell)$ in W', where $\ell \geq \text{lev}(z)$. Since w is contained within z, w is also well-separated with y. Therefore, $(\{w, y\}, \ell)$ exists in W. Conversely, consider $(\{w, y\}, \ell)$ in W, where $\ell \geq \text{lev}(z)$. If there exists a node whose level is less than or equal to ℓ, but higher than w, then findWSPD would have generated a well-separated pair involving it instead of w. Therefore, $(\{w, y\}, \ell)$ belongs as

$(\{z, y\}, \ell)$ in W'. Line 7 in Algorithm 11 changes each such pair to $(\{z, y\}, \ell)$. (Note that the other well-separated pairs related to w whose maximum separation level is less than $\text{lev}(z)$ are related only with w, and they are included in both W and W'.)

Second, consider any well-separated pair, $(\{x, y\}, \ell)$ involving x in W' (Case 1). Such x and y were generated by invoking findDescendants (Line 6 and 7 of Algorithm 10) on nodes u and v of T' that are 1-well separated, respectively. By the nature of the algorithm, u and v are maximally homogeneously 1-well separated. Recall that B is the set of boxes adjacent to $z^{\uparrow\sigma}$ including $z^{\uparrow\sigma}$ in Line 9 of Algorithm 11. We will show that v is contained in some box $b \in B$ (see Fig. 5.4). Assuming this for now, if we let v' be the largest node containing v at level not more than $\text{lev}(z^{\uparrow\sigma})$, for such a b, node(b) in Line 10 of Algorithm 11 will return v'. In the same line, findWSPD(x, v', s) will first find homogeneous 1-well separated pairs involving x and descendants of v', and then invoke findDescendants on each. We are interested in these calls involving ancestors of v. By the maximality of u and v, we assert that no proper ancestor of v will be homogeneously 1-well separated with respect to x. To see this, suppose that there existed a proper ancestor v, denoted by v'' that was homogeneously 1-well separated with x. By definition, a box x' on the same level as v'', containing x is 1-well separated with v''. Since v'' is an ancestor of v, by Lemma 5.3.1(i), $\text{lev}(u) \leq \text{msl}_1(u, v) < \text{lev}(\text{par}(v)) \leq \text{lev}(v'')$. This implies that $\text{lev}(u) < \text{lev}(x')$, and therefore, u is contained in x'. This yields that u and v'' are also homogeneously 1-well separated. This would contradict the fact that u and v are maximally well-separated. Therefore, x is not homogeneously 1-well separated with any proper ancestor of v. Thus, findWSPD(x, v', s) will find the 1-well

separated pair $\{x, v\}$, and will generate the desired s-well separated pair $\{x, y\}$ by invoking findDescendants.

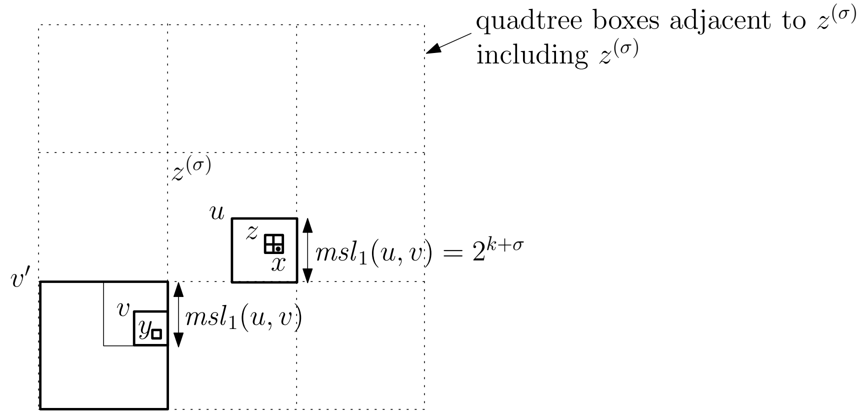


Fig. 5.4: A well-separated pair of new leaf x

To complete the proof, it remains to show that v is contained in some box $b \in B$. To do this, it suffices to show that v is not homogeneously 1-well separated with $z^{\uparrow\sigma}$, and $\text{lev}(v) \leq \text{lev}(z^{\uparrow\sigma})$. By Lemma 5.3.1(ii), $\ell < \text{lev}(z)$. Thus,

$$\text{lev}(u) \;\leq\; \text{msl}_1(u, v) \;=\; \ell + \sigma \;<\; \text{lev}(z) + \sigma \;=\; \text{lev}(z^{\uparrow\sigma}).$$

By the maximality of u and v, v is not homogeneously 1-well separated with any box containing u on a higher level than $\text{msl}_1(u, v)$. Clearly, $z^{\uparrow\sigma}$ is such a box. Also, we have $\text{lev}(v) \leq \text{msl}_1(u, v) < \text{lev}(z^{\uparrow\sigma})$. This completes the proof.

□

5.4.2 Deletion

Deletion essentially is the reverse of the insertion procedure, and is presented in Algorithm 12. Let x be the leaf containing the deleted point, p. If its parent, z has

only two nodes, x and w, the node z will be deleted from the compressed quadtree. Thus, in a symmetrical manner with insertion, we associate the well-separated pairs with w, instead of z (see Line 5). Also, all well-separated pairs associated with x are deleted (see Line 7). The correctness proof is symmetrical with the proof for insertion and is omitted.

Algorithm 12 Deletion of a point p for output sensitive WSPD

Input: A point, p and a separation factor, $s \geq 1$

```
1: function delete(p, s)
2:     Find a leaf x including p
3:     z : parent of x
4:     if z has two nodes, x and x's sibling, w then
5:         Change ({z, v}, ℓ) to ({w, v}, ℓ).
6:         Delete z
7:     Delete all well-separated pairs related to x
8:     Delete x
```

5.4.3 Maintenance of WSPD for updates

In order to analyze the running time of our update algorithms, we need to see how to maintain the s-well separated pairs and the maximum separation level associated with each.

We store all the well-separated pairs in a hash table, denoted by H. Given a box u', let $W(u')$ denote the set of all well-separated pairs (u, v) such that u' is the box at level $\mathrm{msl}_s^*(u, v)$ containing u. Note that a box may be uniquely identified by the Morton code of its lower left corner and its depth. We combine these to save as the key of the hash table. For a box u', let $H(u')$ denote the hash table entry associated with u'. The entry, $H(u')$ stores a linked list of the elements of $W(u')$.

The element of this linked list associated with (u, v) stores a cross reference to an element of the list in $H(v')$, where v' is the box at level $\mathrm{msl}^*_s(u, v)$ containing v. Note that a well-separated pair (u, v) is stored as the pair (u', v'). In addition, we store a bit for each entry in the hash, which we call the *dirty bit.* If $H(u')$ is empty, this bit is set. This will be used for lazy deletion, which will be described below. In this way, the space of the hash table is proportional to the number of well-separated pairs.

For each node u, let L_u denote the set of levels ℓ, such that there exists a well-separated pair (u, v) such that $\mathrm{msl}^*_s(u, v) = \ell$. We assume that L_u is stored as a bit vector indexed by levels. If there exists a well-separated pair involving u at level ℓ, bit ℓ of L_u is set. Through this bit vector, we can compute boxes u' associated with u in the manner presented above. Note that given u and such a level number ℓ, it is possible to compute u' and its associated hash key in $O(1)$ time. This key is then used to access the hash table. Recall that each point's coordinates are stored as fixed-point binary numbers, and bitwise operations can be performed on them in $O(1)$ time. We assume that each coordinate can be stored in a constant number of words, and hence the total space required for the coordinates is proportional to the number of nodes in the tree.

Lemma 5.4.2 *Assume we have a model of computation that supports in $O(1)$ time the bitwise operations boolean-and ("$\wedge$"), boolean-or ("$\vee$"), complement ("$\overline{x}$"), left shift ("$\ll$"), and integer base-2 logarithm (that is, the index of the most significant bit) on point coordinates. Through the bitwise operations, the following operations*

for bit vectors can be performed in $O(1)$ time.

- $\text{add}(L_u, \ell)$*: Add bit ℓ to L_u.*

- $\text{remove}(L_u, \ell)$*: Remove bit ℓ from L_u.*

- $\text{isMember}(L_u, \ell)$*: Return true if L_u has bit ℓ. Otherwise, return false.*

- $\text{split}(L_u, \ell, L_v)$*: Split the bit vector L_u into two bit vectors, such that one contains bits whose positions are greater than or equal to ℓ, and the other contains those that are less than ℓ. The higher bits are stored in a new bit vector L_v, and the lower entries remain in L_u.*

- $\text{merge}(L_u, L_v)$*: Merge bits of L_u to L_v.*

- $\text{getMembers}(L_u)$*: Enumerate all bits in L_u.*

Proof:

- $\text{add}(L_u, \ell)$: Set the bit at ℓ by taking bitwise-or of L_u and the bit string x only whose bit at ℓ is set.

- $\text{remove}(L_u, \ell)$: Reset the bit at ℓ by making out the bit at ℓ, which can be implemented by taking bitwise-and of L_u and $\overline{x}$, where x is the bit string only whose bit at ℓ is set.

- $\text{isMember}(L_u, \ell)$: Return the result of bitwise-and of L_u and the bit string x only whose bit at ℓ is set.

- $\text{split}(L_u, \ell, L_v)$: We do this by first copying L_u to L_v. We then mask out all bits below level ℓ from L_v and mask out all bits at or above ℓ from L_u.

- merge(L_u, L_v): Merge two bit vectors L_u to L_v by taking bitwise-or of them.
- getMembers(L_u): First copy L_u to the temporary bit string x. Output integer base-2 logarithm of x, which is the index of x's most significant bit, and then this bit from x by the operation remove. Repeat this until x have no more bit that is set.

Above operations except the operation getMembers take $O(1)$ time, and the operation getMembers takes $O(|L_u|)$ time. Letting p is the representative point of u, the size of a bit vector, L_u is less than the size of p's coordinates, and hence $O(|L_u|) = O(1)$. Thus, all of above operations can be performed in $O(1)$. □

Let us consider how to implement point insertion, deletion, and node queries.

Point Insertion: As mentioned above, the first step is to insert the new point p into the compressed quadtree. Following Fischer and Har-Peled [48] this can be performed in $O(\log n)$ time by representing the compressed quadtree (which may be unbalanced) as a topology tree [50].

To bound the time needed to update the WSPD, recall nodes x, z, and w from Algorithm 11. If x's parent, z, is new (Case 2), we update the well-separated pairs involving x's sibling, w. That is, each well-separated pair $(\{w, v\}, \ell)$, where $\ell \geq \text{lev}(z)$ is changed to $(\{z, v\}, \ell)$. We do this by performing split($L_w, \text{lev}(z), L_z$). This can be done in $O(1)$ time.

Next, we find all the new well-separated pairs involving the leaf x. We first find the set B of boxes at level $\text{lev}(z^{\uparrow\sigma})$ that are not homogeneously 1-well separated

with respect to $z^{\uparrow\sigma}$ (see Line 9 of Algorithm 11). Observe that because we assume the L_∞ metric, B consists of boxes of side length equal to the side length of $z^{\uparrow\sigma}$. There are at most 3^d such boxes, and they can be computed in $O(1)$ time. For each $b \in B$, we invoke node(b) to compute the highest level node of the compressed quadtree contained within b (see Line 10). This query can be performed in $O(\log n)$ time through the use of the topology tree. Then, we invoke findWSPD(x, node(b), s). For each newly generated pair $\{x, y\}$ resulting from findWSPD, we first update both L_x and L_y by add(L_x, $\mathrm{msl}^*_s(x, y)$), and add(L_y, $\mathrm{msl}^*_s(x, y)$). These take $O(1)$ time. Let x' and y' be boxes on level $\mathrm{msl}^*_s(x, y)$ containing x and y, respectively. We add elements referencing this well-separated pair into both $H(x')$ and $H(y')$. The dirty bits of both $H(x')$ and $H(y')$ are then reset. This can also be performed in $O(1)$ time. Thus, including the $O(\log n)$ time to insert the point, the addition of all new well-separated pairs involving x can be performed in worst-case time $O(\log n + m)$, where m is the number of newly generated pairs.

Point Deletion: Next, let us consider the deletion of a point p. We first find the leaf x containing p (see Line 2 of Algorithm 12). This can be performed in $O(\log n)$ time, again using the topology tree. Recall z and w from the algorithm. If z would have only one child w after the deletion of x, z will be also deleted. We then merge all the well-separated pairs involving z to w by merge(L_z, L_w) in $O(1)$ time.

Next, we remove all well-separated pairs involving x. First, we get all bits of L_x that are set by getMembers(L_x). This takes $O(1)$ time. For each bit ℓ of L_x that is set, let x' denote the box of level ℓ containing x. We access the linked

list in $H(x')$. If the dirty bit of $H(x')$ is not set, that is, $H(x')$ has a nonempty list, we remove all the elements of the list. For each associated pair (x', y'), we also remove the corresponding element from $H(y')$. If $H(y')$ is now empty, we set its dirty bit. If we attempt to access $H(y')$ as part of a future operation, it is because the corresponding bit of L_y was set, for some node y. Thus, whenever we access any entry $H(y')$, if its dirty bit is set, we reset the corresponding bit of L_y as well as the dirty bit. By charging this to the earlier deletion that set the dirty bit, we can easily see that this takes the $O(1)$ amortized time. After removing all the elements of the list in $H(x')$, we remove ℓ from L_x by remove(L_x, ℓ) in $O(1)$ time. If the dirty bit of $H(x')$ is set, we remove the corresponding bit ℓ from L_x, and reset the dirty bit like above in $O(1)$ amortized time. Combining this with the initial $O(\log n)$ time to remove the point, deletion can be performed in amortized time $O(\log n + m)$, where m is the number of or deleted pairs.

Combining these results, we have Theorem 5.1.1.

Chapter 6

Maintaining Nets and Net Trees under Incremental Motion[1]

6.1 Introduction

Up to this point in the dissertation, we have interpreted the notion of "dynamic data structures" in the classical sense, meaning allowing the insertion and deletion of individual points. In many applications, however, the notion of dynamics involves points that move over time. In this chapter, we will explore the design of data structures for dealing with dynamics in this form.

The problem of maintaining geometric structures for points in motion has been well studied over the years in computational geometry. The vast majority of theoretical work in this area has been based on the assumption that point motion is continuous and the future motions of the points are known in advance [59]. In practice, however, motion is typically presented incrementally in discrete time steps and the long-term motion of points is not known and may not even be predictable. In this chapter, we present efficient online algorithms for maintaining a data structure for a point set undergoing incremental motion.

For the underlying data structure, we consider the maintenance of nets and

[1]The material appearing in the chapter is based on the paper "Maintaining Nets and Net Trees under Incremental Motion" [29].

net trees. Let P denote a finite set of points in some metric space $\mathcal{M}$, which may be either continuous (as in Euclidean space) or discrete. Given $r > 0$, an *r-net* for P is a subset $X \subseteq P$ such that every point of P lies within distance r of some point X, and no two points of X are closer than r. Each point of P can be associated with a covering point of X that lies within distance r, which is called its *representative*. We can easily derive a tree structure, by building a series of nets with exponentially increasing radius values, and associating each point at level $i-1$ with its representative as parent at level i. Like the vantage point tree (Vp-tree) [115] and the generalized hyperplane tree (Gh-tree) [108], the net tree can be viewed as a metric generalization of hierarchical partition trees like quadtrees [94]. (Formal definitions of nets and net trees are given in Section 6.2.)

Our algorithm is allowed some additional slackness in the properties of the net to be maintained. For example, while the optimal algorithm is required to maintain all points within distance r of each net point, we allow our algorithm for nets to maintain a covering distance of $2r$ for nets and $4r$ for net trees. (See Section 6.2 for the exact slackness conditions.) Because our principal motivation is in maintaining net trees under motion, we impose the assumption that the input points to our r-net algorithm arise from an $(r/2)$-net.

We establish the efficiency of our online algorithms by providing an upper bound on the *competitive ratio* on the communication cost of our algorithm, that is, the worst-case ratio between the communication costs of our algorithm (subject to the slackness conditions) and any other algorithm (without the slackness). The exact results are presented in Sections 6.3 and 6.4. Assuming that the points are

in a space of constant doubling dimension (e.g., Euclidean of constant dimension), we achieve a competitive ratio of $O(1)$ for the maintenance of a net and $O(\log^2 \Phi)$ for the net tree, where Φ is the aspect ratio of the point set (the ratio between the maximum and minimum interpoint distances). Our online algorithm makes no *a priori* assumptions about the motion of the points. The competitive ratio applies even if the optimal algorithm has full knowledge of point motion, and it may even have access to unlimited computational resources. The constant factors hidden by the asymptotic notation grow exponentially in the doubling dimension of the space.

The rest of the chapter is organized as follows. In the next section, we present definitions and background information. In Section 6.3, we present our algorithm for maintaining a net, and in Section 6.4, we present our algorithm for maintaining a net tree.

6.2 Preliminaries

We begin with some basic definitions, which will be used throughout the paper. Let $\mathcal{M}$ denote a metric space, with associated distance function dist: $\mathcal{M} \times \mathcal{M} \to \mathbb{R}$. (This means that dist is symmetric, positive definite, and satisfies the triangle inequality.) Throughout, we let P be a finite subset of points in some metric space $\mathcal{M}$. For a point $p \in \mathcal{M}$ and a real $r \in \mathbb{R}^+$, let $b(p, r) = \{q \in \mathcal{M} : \mathrm{dist}(p, q) < r\}$ denote the open ball of radius r centered at p. The *doubling constant* of the metric space is defined to be the minimum value λ such that every ball b in $\mathcal{M}$ can be covered by at most λ balls of at most half the radius. The *doubling dimension* of

the metric space is defined as $d = \log_2 \lambda$. Throughout, we assume that $\mathcal{M}$ is a space of constant doubling dimension.

Recall that P is a finite set of points in some metric space $\mathcal{M}$. Given $r \in \mathbb{R}^+$, an *r-net* for P [64] is a subset $X \subseteq P$ such that, for some constant $\gamma \geq 1$,

$$\max_{p \in \mathcal{M}} \mathrm{dist}(p, X) \ < \ r \qquad \text{and} \qquad \min_{\substack{x,y \in X \\ x \neq y}} \mathrm{dist}(x, y) \ \geq \ \frac{r}{\gamma}.$$

The first constraint is called the *covering constraint*, and the second is called the *packing constraint*. Intuitively, a net defines a simple type of clustering of the point set into balls of radius r, and each point $p \in P$ can be associated with a *representative* $x \in X$ (denoted by $\mathrm{rep}(p)$) lying within distance r. No two representatives are closer than r. Note that the representative is not necessarily unique. To simplify notation, henceforth, we assume that $\gamma = 1$, but our results apply to any constant γ.

In order to establish our competitive ratio, we will need to relax the r-net definition slightly. Given constants $\alpha, \beta \geq 1$, an *(α, β)-slack r-net* is a subset $X \subseteq P$ of points such that,

$$\max_{p \in \mathcal{M}} \mathrm{dist}(p, X) \ < \ \alpha\, r \qquad \text{and} \qquad \forall x \in X, \ |\{X \cap b(x, r)\}| \ \leq \ \beta.$$

Thus, we allow each point to be farther from the closest net point by a factor of α, and we allow net points to be arbitrarily close to each other, but there cannot be more than β points within distance r of any net point. Clearly, an (α, β)-slack r-net is an (α', β')-slack r-net for $\alpha' \geq \alpha$ and $\beta' \geq \beta$. When we wish to make

the distinction clearer, we will use the term *strict r-net* to denote the standard definition, which arises as a special case when $\alpha = \beta = 1$.

Before introducing net trees, we first introduce the concept of the *aspect ratio* (or *spread*) of a static point set to be the ratio of the diameter of P and the distance between the closest pair of points in P. Since we will be dealing with points in motion, we assume that we are two values δ and Δ, which provide a lower and upper bound, respectively, on the distances between any two points of P throughout the course of the motion. We define $\Phi(P)$ to be Δ/δ. By scaling distances, we may assume that $\delta = 1$.

A *net tree* of P is defined as follows. The leaves of the tree consists of the points of P. Note that by our assumption that $\delta = 1$, these form a 1-net of P itself, which we denote by $P^{(0)} = P$. The tree is based on a series of nets, $P^{(1)}, P^{(2)}, \ldots, P^{(m)}$, where $m = \lceil \log_2 \Phi \rceil$, and $P^{(i)}$ is a (2^i)-net for $P^{(i-1)}$. Observe that $|P^{(m)}| = 1$. (More generally, we may replace 2 with any constant that is greater than 1.) Recall that, for each $p \in P^{(i-1)}$, there is a point $x \in P^{(i)}$, called its representative, such that $\mathrm{dist}(p, x) \leq 2^i$. We declare this point to be a *parent* of p, which (together with the fact that $|P^{(m)}| = 1$) implies that the resulting structure is a rooted tree. An easy consequence of the packing and covering constraints is that the number of children of any node of this tree is a constant (depending on the doubling constant of the containing metric space). Our usage of the term "net tree" is not standard. There have been a number of related data structures, based on a hierarchical collection of nets. Our definition is based on the simplest forms of net tree [52, 75], whose height depends on the aspect ratio. This is in contrast to more sophisticated forms given

in [33, 55, 64], which achieve height of $O(\log |P|)$, independent of the aspect ratio.

This definition can be easily generalized to assume that the nets forming each level of the tree are (α, β)-slack nets. We refer to such a tree as an (α, β)-slack net tree. Assuming that α and β are constants, this relaxation will affect only the constant factors in the asymptotic complexity bounds.

Because our ultimate interest is in maintaining net trees under incremental motion, it will be convenient to impose an additional constraint on the points P. In a net tree, the input to the ith level of the tree is a (2^{i-1})-net, from which we are to compute a 2^i net. Thus, in our computation of an r-net, we will assume that the point set P is an $(r/2)$-net. In addition to just moving the points, we will also allow points to be inserted or deleted from the set at any time.

Recall that we interested in maintaining points under incremental black-box motion. More formally, we assume that the points change locations synchronously at discrete time steps $T = \{0, 1, \ldots, t_{\max}\}$. Given a point $p \in P$ and $t \in T$, we use p to refer to the point in the symbolic sense, and (when time is significant) we use p_t to denote its position at time t.

Let us now consider the certificates used in the maintenance of an r-net. In order to maintain an (α, β)-slack r-net, the observer must be provided enough information to verify that the covering and packing constraints are satisfied. At any time t, let P_t denote the current point set and let X_t denote the current slack net. We assume the incremental maintenance of any net is based on the following two types of certificates, where the former validates the covering constraint and the latter validates the packing constraint.

Assignment Certificate: Given $p \in P$ and $x \in X$, $\mathrm{rep}(p) = x$, and therefore at each time t, $\mathrm{dist}(p_t, x_t) < \alpha r$.

Packing Certificate: Given $x \in X$, at each time t we have $|X_t \cap b(x_t, r)| \leq \beta$.

The first condition requires constant time to verify. The second condition involves answering a spherical range counting query. For the purposes of the results presented here, it suffices to answer such queries approximately to within a constant approximation error (which only affects the constant factors in the analysis). Observe that $O(|P|)$ certificates suffice to maintain the net.

6.3 Incremental Maintenance of a Slack Net

In this section we present an online algorithm for maintaining an r-net for a set of points undergoing incremental motion and provide an analysis of its competitive ratio. Given our metric space $\mathcal{M}$, let $\beta = \beta(\mathcal{M})$ denote the maximum number of balls of radius r that can overlap an arbitrary ball of radius r, such that the centers of these balls are at distance at least r from each other. We shall show in Lemma 6.3.3 below that $\beta \leq \lambda^2 = 4^d$, where λ is the doubling constant of $\mathcal{M}$, and d is the doubling dimension of $\mathcal{M}$. The main result of this section is as follows.

Theorem 6.3.1 *Consider any metric space $\mathcal{M}$ of constant doubling dimension, and let $\beta = \beta(\mathcal{M})$ be as defined above. There exists an incremental online algorithm, which for any real $r > 0$, maintains a $(2, \beta)$-slack r-net for any point set P under incremental motion in $\mathcal{M}$. Under the assumption that P is a $(2, \beta)$-slack $(r/2)$-net,*

the algorithm achieves a competitive ratio of at most $(\beta\lambda^3 + 2)(\beta + 2) = O(\lambda^7) = O(1)$.

The remainder of this section is devoted to proving this theorem.

6.3.1 Online Algorithm for Maintaining a Slack Net

In this section we present our online algorithm. The algorithm begins by inputting the initial placements of the points, and it communicates an initial set of certificates to the observer. (We will discuss how this is done below.) Recall that the observer then monitors the point motions over time, until first arriving at a time step t when one or more of these certificates is violated or when a point of P is explicitly inserted or deleted. It then wakes up the builder and informs it of the current event. The builder applies the operations as described in our algorithm below, and returns control to the observer.

Our algorithm maintains not only the points of the slack r-net, which we denote by X, but also the assignment of each point $p \in P$ to its representative $\mathrm{rep}(p) \in X$. For each point $p \in P$, we maintain a subset $\mathrm{cand}(p) \subseteq X$, called the *candidate list*.

Before describing the incremental update process, we begin with three useful utility operations: *reassignment*, *net-point creation*, and *net-point removal*.

Reassign(p): If $\mathrm{cand}(p) \neq \emptyset$, repeatedly extract elements from $\mathrm{cand}(p)$ until finding a candidate $x \in X$ that lies within distance $2r$ of p. If such a candidate is found, set $\mathrm{rep}(p) \leftarrow x$, and create a new assignment certificate involving p and

x. If no such candidate exists, invoke the net-point creation operation for p.

Create net point(p): We assume the precondition that cand(p) $= \emptyset$. Add p to the current net X. Set rep(p) $\leftarrow p$. For each point $p' \in P \setminus \{p\}$ such that dist(p', p) $< 2r$, add p to the candidate list cand(p'). Finally, create a packing certificate for p.

Remove net point(x): First, x is removed from both X and all the candidate lists that contain it. Remove any packing certificate involving x. For all $p \in P$ such that rep(p) $= x$, invoke the reassignment operations on p.

Note that the reassignment operation generates one new assignment certificate (for p) and may create one packing certificate if p is added to X. Let us now consider the possible actions of the builder, once the observer reports an event (point insertion or deletion) or a certificate violation (assignment or packing).

Insert point(p): Set cand(p) to be the set of net points $x \in X$ such that dist(p, x) $< 2r$. Then apply the reassignment operation to p.

Delete point(p): All certificates involving p are removed. If $p \in X$, then invoke the net-point removal operation on p. Finally, remove p from P.

Assignment-certificate violation(p): Let p be the point involved, and let $x =$ rep(p) be its representative. Remove x from p's candidate list and apply the reassignment operation to p.

Packing-certificate violation(x): Invoke net-point removal for each point of $X \cap b(x, r)$.

Observe that the processing of any of the above events results in a constant number of changes to the certificate set, and hence in order to account for the total communication complexity, it suffices to count the number of operations performed.

Initially, X is the empty set, and we start the process off by invoking the insertion operation for each $p \in P$, placing it at its starting location. Observe that after the processing of each assignment- and packing-certificate violation, the condition causing the violation has been eliminated, and therefore we have the following.

Lemma 6.3.1 *If no certificates are currently violated, the set X maintained by the above algorithm is a valid $(2, \beta)$-slack r-net.*

6.3.2 Competitive Analysis for Maintaining Nets

In this section we present a competitive analysis of the computational complexity of the online algorithm presented in the previous section. Recall that P denotes the point set, which is in motion of some finite time period. For any time t, let $N_t(o)$ denote the *optimal neighborhood* consisting of the points of P that have o assigned as their representative by the optimal algorithm. Our competitive analysis is based on showing that each of the operations performed by our algorithm can be charged to some operation of the optimal algorithm, in such a manner that each optimal operation is charged a constant number of times (depending on the doubling dimension and associated packing lemma).

First, we present some geometric preliminaries, which will be useful later in

the analysis. Recall that λ denotes the doubling constant of the metric space.

Lemma 6.3.2 *Given any (α, β)-slack r-net X, at most $\beta\lambda^{\lg\lceil 2R/r\rceil}$ points of X can lie within any ball of radius R.*

Proof: First, we claim that for any p in the metric space, $b(p, r/2)$ contains at most β points of X. To see this, suppose not. Take any $q \in b(q, r/2) \cap X$. Clearly, $b(q, r)$ contains $b(p, r/2)$, and hence would contain more than β points contradicting the fact that it is an (α, β)-slack r-net. In a metric space with doubling constant λ, any ball of radius R can be covered by at most $\lambda^{\lg\lceil R/(r/2)\rceil} = \lambda^{\lg\lceil 2R/r\rceil}$ balls of radius $r/2$. Thus, there are at most $\beta\lambda^{\lg\lceil 2R/r\rceil}$ points in X within any ball of radius R. □

Recall that the points of our slack r-net are assumed to arise from a slack $(r/2)$-net. By the above lemma and the definition of (α, β)-slack r-net, we have the following.

Corollary 6.3.1 *Let P be an (α, β)-slack $(r/2)$-net and let X be an (α, β)-slack r-net for P. Then the number of points of X (respectively, P) that lie within a ball of radius $2^k r$ is at most $\beta\lambda^{k+1}$ (respectively, $\beta\lambda^{k+2}$).*

Our choice of $\beta = \lambda^2$ in Theorem 6.3.1 is a direct consequence of the following lemma.

Lemma 6.3.3 *Let Z be a set of balls of radius r whose centers are taken from a (strict) r-net. Then any ball b of radius r (not necessarily in Z) can have a nonempty intersection with at most λ^2 balls of Z.*

Proof: Let b' be a ball concentric with b of radius $2r$. Each ball of Z overlapping b has its center inside b'. Let X be the set of centers of balls in Z. Since all the points of X are separated from each other by distance r, any ball of radius r centered at a point of X contains at most one point of X. (Recall that balls are open.) Thus, by applying Lemma 6.3.2 (with $\beta = 1$), there can be at most $\lambda^{\lg\lceil 2 \cdot 2r/r \rceil} = \lambda^2$ points of X within a ball of radius $2r$. $\square$

Given the motion sequence for the point set P, let n denote the total number of operations performed by our online slack-net algorithm, and let n^* denote the total number of operations processed by any correct (e.g., the optimal) algorithm. In order to establish the competitive ratio, it suffices to show

$$n \;\leq\; (\beta\lambda^3 + 2)(\beta + 2)n^*. \tag{6.1}$$

The remainder of this section is devoted to showing this.

Our analysis is based on a charging argument, which relates the total number of slack-net operations to the number of slack-net creations and then relates the number of slack-net creations to the number of optimal operations. Let n^*_A, n^*_C, n^*_R, n^*_I, and n^*_D, and denote, respectively, the total number of assignments, net point creations, net-point removals, point insertions, and point deletions performed by the optimal algorithm. Let n_A, n_C, n_R, n_I, and n_D denote corresponding quantities for our slack-net algorithm. Thus, we have

$$n \;=\; n_A + n_C + n_R + n_I + n_D.$$

First, we bound the total number of assignments in terms of the number of point insertions and slack-net creations. Changes in assignment in our algorithm occur as a result of running of the reassignment operator. Since the assignment is made to some point of the candidate list, it suffices to bound the total number of insertions into candidate lists. This occurs when points are inserted and when net points are created.

Lemma 6.3.4 $n_A \leq \beta\lambda^3 n_C + \beta\lambda^2 n_I$.

Proof: Observe that since each reassignment is made to a point in a candidate list, n_A is bounded by the total number of additions to all the candidate lists throughout the course of the algorithm. Such an addition may occur whenever either: (a) a net point is created or (b) a new point is inserted. Whenever a net point x is created, it is added to the candidate lists of all points within distance $2r$. Whenever a point p is inserted it adds all the net points within distance $2r$ of itself to its candidate list. By Corollary 6.3.1, the number of such candidate-list additions is at most $\beta\lambda^3$ and $\beta\lambda^2$, respectively. □

Since point insertions and deletions must be handled by any correct algorithm, we have $n_I = n_I^*$ and $n_D = n_D^*$. The total number of net point removals (n_R) cannot exceed the total number of net point creations (n_C). Thus, it suffices to bound n_C, the total number of slack-net point creations.

Before bounding n_C we make a useful observation. Whenever a net point x is created, it adds itself to the candidate list of all points that lie within distance $2r$. Since (by strictness) the diameter of any optimal neighborhood is at most $2r$,

it follows that, in the absence of other events, the creation of net-point x within an optimal neighborhood inhibits the creation of any other net points within this neighborhood. Given $t \leq t'$, let $(t, t']$ denote the interval $[t + 1, t']$.

Lemma 6.3.5 *Let o denote an optimal net point. Suppose that no optimal assignments occur to the points of $N(o)$ during the time interval $(t, t']$, $x \in N(o)$ is added to X at time t, and x is not removed from X throughout $(t, t']$. Then, for any time in $(t, t']$ no point $p \in N(o)$ will be added to X.*

Proof: When x is added to X at time t, x is added to the candidate lists of all the points that lie within distance $2r$ of x, which includes all the points that are currently in $N(o)$. During the interval $(t, t']$, no new points are added or removed from $N(o)$, and x is not removed. Thus, x is a valid representative for each $p \in N(o)$. Therefore, (by our reassignment operator) no point of $N(o)$ will be added to X during this time interval. □

This implies that, without optimal assignments, each optimal net point o can have at most one corresponding slack net point $x \in N(o)$. Furthermore, if x is removed from X (e.g., as the result of a packing-certificate violation), only one of the points of $N(o)$ may replace it as a slack-net point. When a net point within an optimal neighborhood is created to replace a removed net point, we call this a *recovery*. Whenever a packing-certification violation occurs, at least $\beta + 1$ slack-net points are removed. The following lemma implies that the number of recovered net points is strictly smaller.

Lemma 6.3.6 *The number of net points recovered as a result of the processing of a packing-certificate violation is at most* β*.*

Proof: Whenever a packing-certificate violation occurs at some point x, all the points of X within distance r of x are removed. Consider the set O' of optimal centers whose neighborhoods contain one of these removed net points. Consider the ball of radius r centered at each point of O'. By the properties of a strict net and Lemma 6.3.2, $|O'| \leq \beta$. By Lemma 6.3.5, at most one slack net point can be created within each of these optimal neighborhoods. Therefore, we obtain the desired bound. $\square$

We say that an optimal neighborhood $N(o)$ is *crowded* (at some time t) if $|N_t(o) \cap X_t| \geq 2$. Our next lemma states that whenever a packing-certificate violation occurs, at least two of removed net points lie in the same crowded neighborhood.

Lemma 6.3.7 *Consider a packing-certificate violation which occurs in the slack net but not within the optimal net, and let* $X' \subseteq X$ *denote the net points that have been removed as a result of its handling. Let* O' *denote a subset of* O *of overlapping neighborhoods, that is,* $O' = \{o \mid N(o) \cap X'\}$*. Then, there exists at least one optimal center* $o \in O'$ *such that* $|N(o) \cap X'| \geq 2$*.*

Proof: We can assume that no optimal packing-certificate violation occurs since optimal algorithm can avoid the packing-certificate violation simply by using optimal assignments. Thus, assume that O' is consistent during our operation.

A packing-certificate violation is triggered by at least $\beta + 1$ slack-net points lying within in a ball $b(x, r)$ for some $x \in X$, and $X' = X \cap b(x, r)$. For each $o \in O'$,

$N(o) \cap X'$ consists of some subset of points of X' that lie within the ball $b(o, r)$. By Lemma 6.3.3, at most β balls from O can have a nonempty intersection with $b(x, r)$. Clearly, therefore, at least one of these subsets contains at least two points of X'.

□

The handling of the packing-certificate violation removes the points of X', and by Lemma 6.3.5, this optimal neighborhood will recover at most one net point. Thus, in the absence of other effects (optimal reassignment in particular) the overall crowdedness of the system strictly decreases after processing each packing-certificate violation. Intuitively, crowdedness increases whenever a point of the slack net is moved from one optimal neighborhood to another. But such an event implies that the optimal algorithm has changed an assignment. We can therefore charge slack-net point creations to optimal assignments. The following lemma formalizes this intuition.

Lemma 6.3.8 *Each net point creation can be uniquely charged an optimal operation.*

Proof: Consider the creation of a slack-net point x at some time t and let o be the optimal net point such that $x \in N(o)$. We consider two cases. First, there exists $x' \in X$ such that $x, x' \in N(o)$ and $x' \in \text{cand}(x)$ at some time $t' < t$. (This is a case of recovery.) Second, no such x' exists. This can be further divided into two subcases. First, x is the first slack-net point to appear in $N(o)$ after o was added to O. Second, some slack-net points in $N(o)$ had already been added to X, but x had never obtained any of them as its candidates. That is, from the time that x was

assigned to $N(o)$ until time t, there had been no slack-net point creations in $N(o)$. More formally, we have the following cases:

1. [Recovery:] Let t' be the latest of the following three events to occur: (1) x is assigned to $N(o)$, (2) x' is assigned to $N(o)$, and (3) x' is added to X. At time t', $x' \in \text{cand}(x)$. Without loss of generality, x' is the last element of $\text{cand}(x) \cap N(o)$. Since x was added to X, we know that x' is not valid at time t. There are three possible reasons. Either:

   (a) x' was assigned to some other optimal center o',

   (b) x' was deleted, or

   (c) x' was removed from X.

2. (a) [Initial Creation:] x is the first slack-net point added to $N(o)$. (Note that this does not imply that there were no slack-net points in $N(o)$, but clearly any such points were not created when they were in $N(o)$.) The creation of x is charged to the creation of o. Clearly, this event can be charged at most once.

   (b) [x Recently Arrived:] Let $t' < t$ be last time such that point $x' \in N(o)$ was added to X and suppose that x was assigned to $N(o)$ at some time $t'' > t'$.

   We observe that the optimal assignment of x generates at most one slack-net point in $N(o)$ (namely, x itself). Before x was added to X, it was not a net point. Thus, x is not charged to any other net creations. Clearly,

each such optimal assignment can be charged by at most one such event.

Let us consider Case 1. Observe that each of the possible cases listed above results in the creation of at most one new net point in $N(o)$ by Lemma 6.3.5. The cost of the creation of x will be charged to the operation that caused x' to become invalid.

Cases 1(a) and 1(b) can be charged directly to an optimal assignment or optimal deletion operation, respectively. Clearly this optimal operation is charged at most once.

Finally, let us consider Case 1(c). Note that the removal of x' is not directly related to any optimal operation. First, we show that it is charged at most once. After adding x to X, x becomes a candidate of all points of $N(o)$. (That is, x takes a role that x' had performed in representing some of the points of $N(o)$, since x was the last element in $\mathrm{cand}(p) \cap N(o)$ for all $p \in N(o)$.) Thus, x' is charged only once in this way. We observe that a removal operation only happens during the handling of a packing-certificate violation. During the handling of each packing-certificate violation, we can create at most β net points by Lemma 6.3.6. Thus, the total number of net point creations due to Case 1(c) is bounded by β times the total number of packing-certificate violations. Since each packing-certificate violation satisfies the conditions of Lemma 6.3.7, we observe that the condition can be made by only optimal assignments (see Cases 1(a) and 2(b)). Thus, the total number of net point creations by Case 1(c) is at most $\beta \cdot n_A^*$. □

We summarize that above analysis to obtain the following bound.

Lemma 6.3.9 $n_C \leq (\beta+2)\ n_A^* + n_C^* + n_D^*$.

Proof: Let n_{C1}, n_{C2a}, and n_{C2b} denote the total numbers of net-point creations arising due to cases 1, 2(a), and 2(b), respectively, as given in the proof of Lemma 6.3.8. Clearly, n_{C2a} is at most n_C^* and n_{C2b} is at most n_A^*. As observed earlier, for n_{C1}, there are three cases. Each case bounds to the number of optimal operations. Thus,

$$n_{C1} = n_A^* + n_D^* + \beta \cdot n_A^*. \quad \text{(By Lemma 6.3.6 and 6.3.8)}$$

Therefore, the total number of slack-net point creations is

$$\begin{aligned} n_C &\leq n_{C1} + n_{C2a} + n_{C2b} \\ &\leq (n_A^* + n_D^* + \beta \cdot n_A^*) + n_C^* + n_A^* \\ &\leq (\beta+2)\ n_A^* + n_C^* + n_D^* \end{aligned}$$

□

The proof of Theorem 6.3.1 follows by combining the observations of this section with Lemmas 6.3.4 and 6.3.9.

6.4 Incremental Maintenance Algorithm for Net Tree

Recall that a net tree is based on a hierarchy of nets of exponentially increasing radius values. In particular, the points at level i are a (r_i)-net of the points at level $i-1$, where $r_i = 2^i$. Recall that the height of net tree is $h = \lceil \lg \Phi(P) \rceil$, where

$\Phi(P)$ is the worst-case aspect ratio of the point set. In this section we present an efficient online algorithm for maintaining a slack net tree for a set of points under incremental motion in a metric space $\mathcal{M}$. The main result of this section is presented below. In contrast to the results of the previous section, the slackness parameter α in the net increases from 2 to 4 and the competitive ratio increases by a factor of $O(\lambda h^2)$. Recall from Section 6.3 that $\beta = \lambda^2$ denotes the maximum number of balls of radius r that can overlap an arbitrary ball of radius r, such that the centers of these balls are at distance at least r from each other.

Theorem 6.4.1 *Consider any metric space $\mathcal{M}$ of constant doubling dimension. There exists an online algorithm, which maintains a $(4, \beta)$-slack net tree for any point set P under incremental motion in $\mathcal{M}$. The algorithm achieves a competitive ratio of at most $(\beta\lambda^4 + \beta\lambda^3 + 4)(\beta + 3)h^2 = O(\lambda^8 h^2) = O(h^2)$.*

In the next section we present the algorithm and in the following section we present the competitive analysis of the algorithm.

6.4.1 Incremental Maintenance Algorithm for Net Tree

Intuitively, our algorithm for maintaining the net tree is based on applying the algorithm of the previous section to maintain the net defining each level of the tree. Creation and removal of net points at level will result in point insertions and deletions at other levels of the tree. Let $O^{(i)}$ and $X^{(i)}$ denote nets at level i of the tree generated by (strict) optimal algorithm and our slack-net algorithm, respectively. Since $X^{(i)}$ is a slack r_i-net of $X^{(i-1)}$, it will be convenient to use $P^{(i)}$

as a pseudonym for $X^{(i-1)}$, in order to maintain symmetry with the terminology of the previous section.

The competitive analysis of the previous section (recovery and crowdedness, in particular) was based on the relationship between slack-net points and neighborhoods of the optimal net. However, except at the leaf level, there is no reason to believe that points of $P^{(i)}$ will reside in level i of the optimal net tree. Consider an optimal net point $o \in O^{(i)}$. We define the optimal neighborhood, still denoted by $N(o)$, to be the set of points of P lying in the leaves of the tree that are descended from o. Because of the exponential decrease in the radius values, it is easy to see that the descendants of o lie within distance of o of $2^i + 2^{i-1} + \cdots + 1 < 2^{i+1} = 2r_i$. Thus, the diameter of $N(o)$ is at most $4r_i$. We have the following result.

Lemma 6.4.1 *Let o denote an optimal net point at level i. Then, for any $x \in N(o)$, $N(o) \subseteq b(x, 4r_i) \cap P$.*

This lemma implies that, if we choose any point $x \in N(o)$ to be in our $(4, \beta)$-slack net, it can be used to cover all the points of the optimal neighborhood. This observation justifies our choice of $\alpha = 4$ in Theorem 6.4.1.

Let us now consider the operations performed by our algorithm at level i. Each operation is defined in terms of the analogous single-net operation of Section 6.3.1 applied to the points at this level of the net tree. In each case the operation may cause events to propagate to higher levels of the tree. We begin by describing a few utility operations. Throughout i denotes the tree level at which the operation is being applied.

Reassign(p, i): Invoke the net-reassignment operator to p for $X^{(i)}$, but use the level-i net-creation operator (rather than the single-net operator).

Create net point(p, i): Invoke the net-point creation operation for p in $X^{(i)}$, with one difference. The point p is added to the candidate lists of points within distance $4r_i$ (rather than $2r_i$). Invoke point insertion for p at level $i + 1$.

Remove net point(x, i): Invoke the net-point removal operation for x in $X^{(i)}$. Invoke a delete-point operation of x at level $i + 1$.

With the aid of these utility operations, we now present the actions taken by the builder in response to the various events.

Insert point(p, i): Invoke the point insertion operation for p in $P^{(i)}$, but with the following change. Set cand(p) to be the set of net points $x \in X^{(i)}$ such that dist$(p, x) < 4r_i$ (rather than $2r_i$).

Delete point(p, i): Invoke the point deletion operation for p in $P^{(i)}$. If $p \in X^{(i)}$, invoke an net-removal operation for p at level i.

Assignment-certificate violation(p, i): Process the assignment-certificate violation for p in $P^{(i)}$, but use the level-i reassignment operation (rather than the single-net reassignment operator).

Packing-certificate violation(x, i): Invoke the packing-certificate processing for x in $X^{(i)}$, but use the level-i net-point removal operation (rather than the single-net removal operation).

Note that operations that involve the creation or removal of net points will result in point insertion or deletion, respectively, at the next higher level of the tree. As with single-net operations, each operation may induce addition certificate violations. All these violations are stored in a priority queue, so that operations are first applied to the lower levels of the tree and then propagate upwards.

6.4.2 Competitive Analysis for Net Tree

We extend the proof of the competitive ratio for maintaining a single net to the case of a net tree. The difference between the proof of net and net tree is that operations in the net tree can propagate events to other levels of the tree. Similar to the analysis of competitive ratio for a net, our analysis is based on relating the number of slack-net operations to the number of slack net-point creation events. Then, we relate net-point creation to a corresponding optimal operation.

In order to establish Theorem 6.4.1, it suffices to show the following, where n denotes the total number of operations executed by our slack-net algorithm and n^* denotes the total number of operations performed by the optimal algorithm.

$$n \;\leq\; (\beta\lambda^4 + \beta\lambda^3 + 4)(\beta + 3)h^2 n^*.$$

As in the single-net analysis, let n_A^*, n_C^*, n_R^*, n_I^*, and n_D^* denote respectively the total number of assignments, net-point creations, net-point removals, point insertions, and point deletions performed by the optimal net-tree algorithm. Let n_A, n_C, n_R, n_I, and n_D denote corresponding quantities for our slack net-tree algorithm.

First, we consider the total number of insertions (n_I) and deletions (n_D). In contrast to the analysis of a single net, the points at level i of the slack net tree may differ from those in the optimal tree, and hence insertions or deletions of points may occur within at various levels within one tree but not the other. Nonetheless, we can show the following bound on the number of these operations.

Lemma 6.4.2 $n_I \leq n_I^* + n_C$ *and* $n_D \leq n_D^* + n_C$.

Proof: Suppose that a point p is inserted at level i of the slack net tree. This implies that p was added to $X^{(i-1)}$. Thus, the number of insertions is at least the number of net point creations. Also, if p is inserted at the leaf level, the optimal algorithm inserts p as well. Thus, the total number of insertions is at most $n_I^* + n_C$.

Similarly, suppose that a point p is deleted from level i. This implies that p is a net point at level $i-1$. If p is deleted from the leaf level, the optimal algorithm deletes p as well. Thus, the total number of deletions is at most $n_D^* + n_C$. □

The following is the analog to Lemma 6.3.4, and its proof is similar.

Lemma 6.4.3 $n_A \leq \beta\lambda^4 n_C + \beta\lambda^3 n_I$.

Proof: Because the radius increase to $4r$, n_A is at most $\beta\lambda^4 n_C + \beta\lambda^3 n_I$ (by Lemma 6.3.4 and Corollary 6.3.1). □

As in the single-net case, the total number of net-point removals (n_R) is bounded by the total number of net-point creations (n_C). It suffices to bound n_C. As we established in Lemma 6.3.5 for the single-net case, any optimal assignment

can generate at most one net-point creation. The same applies to each level of the net tree. This leads to the following observation.

Lemma 6.4.4 *Each optimal assignment is responsible for the creation of at most h net-points in the slack net.*

Proof: Let i denote the highest level of our slack-net tree such that $p \in P^{(i)}$. It follows that $p \in X^{(j)}$, for all $j < i$. Since $p \in X^{(j)}$ may be assigned to another optimal net point, an assignment violation may occur at the points having p as their representative at each such level j. Such points may invoke the reassignment operation and generate a new net point as a result. By Lemma 6.3.5 there will be at most one such recovery per optimal neighborhood. In addition, at level i, p may be added to $X^{(i)}$. Thus, the total number of net points generated by an optimal assignment in a net tree is at most h. □

Recall that an optimal neighborhood is crowded if it contains two or more points of the slack net. As with Lemmas 6.3.6 and 6.3.7 in the single-net case, each packing-certificate violation results in a strict reduction in the number of crowded optimal neighborhoods at this level of the tree.

Lemma 6.4.5 *After handling a packing-certificate violation, the total number of net points lying within crowded optimal neighborhoods decreases by at least one.*

Proof: Let i denote the level at which the packing-certificate violation occurs. By the same procedure as in the single-net case, we invoke the removal operation on all the net points lying within radius r of the affected net point at level i. Recall from

Lemma 6.3.7 that in order for a packing-certificate violation to occur, there exists at least one neighborhood that has at least two net points involved in the violation. By Lemma 6.3.6, the total number of net points of crowded neighborhoods will strictly decrease by at least one at level i.

Let us consider the lower and higher levels of our net tree. This handling operation does not affect any of the levels below i since the net points are removed from $X^{(i)}$ but are still in $P^{(i)}$. At the higher levels, some points will be deleted due to their removal from level i. However, the deletion operation can only reduce the number of net points. Then, during the reassignment operations (as part of the removal operation), some net points may be created by recovery case (1(c)) or creation case (2(a)) of Lemma 6.3.8. However, these do not increase the number of net points in crowded neighborhoods. It then invokes the insertion operation on the next higher level. Whenever a point is inserted, the point searches nearby net points first. It becomes a net point only if there is no net points nearby (that is, within distance $4r$). Thus, the insertion operation does not create a net point in already crowded neighborhoods.

Thus, we have the desired bound. □

This shows that, in the absence of optimal assignments, whenever we process a packing-certificate violation, the total number of net points of crowded neighborhoods strictly smaller. Since each crowding event can be charged to an optimal assignment, we have the following.

Corollary 6.4.1 *The total number of packing-certificate violations is at most* $h \cdot n^*_A$.

Now, we consider all possible reasons that a net-point creation may occur within the slack net tree. The analysis is essentially the same as in Lemma 6.3.8, but there is an increase by a factor of h due to propagation.

Lemma 6.4.6 $n_C \ \leq \ (\beta+3)h^2 \cdot n_A^* + n_C^* + h \cdot n_D^*$.

Proof: Let us consider case 1(c) first. A net point can be removed only as a result of a packing-certificate violation or the explicit deletion of a point. Suppose that at some level, the jth packing-certificate violation occurs. Then, for some k_j, $\beta + k_j$ distinct net points resulted in this violation. Since the handler removes at most $\beta+k_j$ times h net points, by Lemma 6.3.5, at most $(\beta + k_j)h$ net points will be recovered. By Corollary 6.4.1 the sum $\sum_j k_j$ is at most $h \cdot n_A^*$. Thus, the total number of net-point creations due to case 1(c) is at most $\sum_j (\beta + k_j)h \ \leq \ (\beta+1)h^2 \cdot n_A^*$.

For case 2(a), the number of net point creations is clearly at most n_C^*. For cases 1(a), 1(b), and 2(b), the total number of net-point creations is at most $h \cdot n_A^*$, $h \cdot n_D^*$, and $h \cdot n_A^*$, respectively, since each operation at level i can propagate to each of the other h levels of the tree. Thus we have

$$
\begin{aligned}
n_C \quad &\leq \quad (\beta+1)h^2 \cdot n_A^* + n_C^* + h \cdot n_A^* + h \cdot n_D^* + h \cdot n_A^* \\
&\leq \quad (\beta+3)h^2 \cdot n_A^* + n_C^* + h \cdot n_D^*.
\end{aligned}
$$

□

We now summarize the results of this section to obtain the desired competitive ratio.

$$
\begin{aligned}
n &= n_A + n_C + n_R + n_D + n_I \\
&\leq n_A + 2n_C + n_D + n_I \\
&\leq \big(\beta\lambda^3 n_I^* + (\beta\lambda^4 + \beta\lambda^3)n_C\big) + 2n_C + (n_D^* + n_C) + (n_I^* + n_C) \\
&\leq (\beta\lambda^3 + 1)n_I^* + (\beta\lambda^4 + \beta\lambda^3 + 4)\left((\beta + 3)\, h^2 n_A^* + n_C^* + h \cdot n_D^*\right) + n_D^* \\
&\leq (\beta\lambda^4 + \beta\lambda^3 + 4)(\beta + 3)h^2 n^*
\end{aligned}
$$

This completes the proof of Theorem 6.4.1.

Chapter 7

Conclusions

In this dissertation, we have explored the design of data structures for storing dynamic point sets in multidimensional spaces. Our particular focus has been on structures that are useful in applications involving geometric approximations in spaces of relatively low dimension. We have considered dynamics both from the classical perspective of inserting and deleting individual points and in a kinetic context where points move over time. In this chapter, we will summarize the main results of the dissertation and will point to possible directions for future research.

The Quadtreap: In Chapter 3 we introduced a simple dynamic data structure, called the quadtreap, for storing multidimensional point sets. This data structure supports efficient insertion, deletion, and query processing. Because it is possible to update internal node weights efficiently, it is the first dynamic data structure that can answer approximate range counting queries. This data structure is a randomized, balanced variant of a quadtree data structure. The tree is locally restructured through an operation called promotion. When inserted, points are assigned random priorities, and the tree is restructured as if the points had been inserted in priority order.

The quadtreap has logarithmic height (with high probability), which makes it is possible update node weights efficiently. Given a set of n points in $\mathbb{R}^d$, a quadtreap

storing these points has space $O(n)$, and supports insertion in time $O(\log n)$, and deletion in time $O(\log^2 n)$. It also answers approximate range counting queries in time $O\big(\log n + (\frac{1}{\varepsilon})^{d-1}\big)$, when the point weights are drawn from a commutative group.

There are number of interesting topics for future research. One such question is whether the $O(\log^2 n)$ time bound on deletion is really tight, or is it merely an artifact of our analysis. Even if the bound is tight in the worst case, the worst-case involves a particularly bad substructure of parent-child relationships to be present, and thus it cannot affect all deletions. Indeed, we showed that the running time is $O(\log n)$ in expectation if a random point is deleted. This raises the question of whether is possible to modify the tree's definition to achieve $O(\log n)$ deletion time, perhaps by avoiding occurrences of this bad substructure.

Another shortcoming of the data structure is that our best query bounds for approximate range counting rely on the assumption that node weights are drawn from a group, since subtraction of weights is needed. For example, range queries over idempotent semigroups (such as boolean-or or maximum) in which subtraction is not possible, are less efficient. Is it possible to improve the quadtreap's query time for general semigroups?

As we mentioned, the quadtreap is a generalization of the treap data structure of Seidel and Aragon, and therefore it is randomized and its running times hold in expectation (with high probability). Is there a deterministic data structure that supports all essential capabilities of the quadtreap? For example, such a structure might be based on height-balancing (thus generalizing AVL trees) or through rules

for local substructures (thus generalizing red-black trees).

The Splay Quadtree: In Chapter 4 we introduced another variant of a quadtree data structure, called the splay quadtree, which can be thought of as a self-adjusting version of the quadtreap. It can also be thought of as a multidimensional generalization of the well-known (1-dimensional) splay tree. The self-adjusting nature of the data structure means that it can adapt to achieve optimal performance when the access pattern is highly skewed. When a node is accessed, it is brought close to the root through a series of promotions. As a consequence, frequently accessed nodes tend to reside near the root of the tree, and so can be accessed more rapidly.

We proved that the splay quadtree is efficient in amortized sense, meaning the average efficiency of a series of operations. We showed that, given a splay quadtree with n points, the amortized time to splay a node is $O(\log n)$, and the amortized time to insert or delete a point is also $O(\log n)$. We also presented multidimensional generalizations of many of the results of standard splay trees in 1-dimensional space, including the Balance Theorem, the Static Optimality Theorem, the Working Set Theorem, and variants of the Static Finger Theorem for a number of different proximity queries, including box queries (which generalize point-location queries), approximate nearest neighbor queries, and approximate range counting queries.

Standard splay trees are known to satisfy two other important access properties, the Dynamic Finger Property and the Scanning Property. The Dynamic Finger Property states that total access time can be bounded by the sum of logarithms of the distances between consecutively accessed elements. The Scanning

Property states that all items of the tree can be accessed in any order in $O(n)$ time. An important question is whether the splay quadtree also satisfies these access properties.

The quadtreap and splay quadtree are both examples of regular decompositions, which means that the decomposition elements (quadtree boxes here) are fixed, independent of the input point set [94]. This is in contrast to non-regular hierarchical decompositions, such as kd-trees, where the splitting hyperplanes are functions of the point coordinates. Non-regular decompositions have the practical advantage that they can generate tighter approximations to the point set. One direction for future research involves the question of whether there exist data structures that possess the same properties as quadtreaps and splay quadtrees, but are based on non-regular decomposition.

Updating Well-Separated Pair Decompositions: In Chapter 5 we proposed an algorithm for maintaining the well-separated pair decomposition (WSPD) for a dynamic points set, where point insertions and deletions are allowed. Given an n-element point set, this structure concisely represents quadratically many pairs using only $O(n)$ space, subject to a given fixed separation factor. WSPDs have many applications in problems involving distances determined by pairs of points.

Our algorithm allows for insertion and deletion of points into the structure. In particular, these updates are output-sensitive, meaning that the time to insert and delete points depends on the number of changes to the decomposition. Given a dynamic set P, our algorithm maintains a WSPD for P in the L_∞ metric that

supports both insertion and deletion in $O(\log n + m)$ time, where where m denotes the number of newly created (resp., deleted) pairs in the case of insertion (resp., deletion). (The running time is amortized in the case of deletions.) For the sake of efficiency, the specific WSPD structure we maintain is based on a stronger notion of well-separatedness than the standard WSPD. Nonetheless, we show that the size of our strong WSPD is at most a constant factor larger than the size of any standard WSPD.

An obvious question for future research is whether it is possible to maintain WSPDs as efficiently without the need of the stronger notion of separation. WSPDs have numerous applications, for example, computing approximations to the Euclidean minimum spanning tree, constructing geometric spanner graphs, and building distance oracles for spatial networks. For the future research, it would be interesting to see how to incorporate our algorithm into these applications, in order to apply them to dynamic point sets.

Net Trees for Points in Motion: In Chapter 6 we showed how to maintain two fundamental metric-space structures, nets and net trees, for a set of points in motion over time. We assume that motion is presented incrementally over a series of discrete time steps by a black-box, that is, a function that specifies the locations of the points at each time step. We presented a novel model for black-box motion, called the observer-builder model. We presented efficient online algorithms in this model for maintaining the net tree under incremental motion. We established the efficiency of our online algorithms by providing an upper bound on the competitive ratio of

the communication cost of our algorithm, that is, the worst-case ratio between the communication costs of our algorithm and any other algorithm. Assuming that the points are in a space of constant doubling dimension (e.g., Euclidean of constant dimension), we show that our algorithms achieve a competitive ratio of $O(1)$ for the maintenance of a net and $O(\log^2 \Phi)$ for the net tree, where Φ is the aspect ratio of the point set (the ratio between the maximum and minimum interpoint distances).

Our original motivation for proposing this model arose from our interest in producing an efficient implementation of a network embedding problem based on a Markov-chain Monte-Carlo (MCMC) approach. When applied to point sets, MCMC algorithms such the Metropolis-Hastings algorithm can be simulated by a black-box motion model. For example, algorithms for computing latent space embeddings of networks are solved by application of MCMC methods [66]. One possible topic for the future research is to apply our net tree algorithms to the efficient computation of latent space embeddings of networks through the use of MCMC algorithms. Another interesting direction for future work would be to extend our results on nets and net trees to other the maintenance of other proximity structures in the black-box model, such as WSPDs, minimum spanning trees, and Delaunay triangulations.

Bibliography

[1] B. Adams, M. Pauly, R. Keiser, and L. J. Guibas. Adaptively sampled particle fluids. In *ACM SIGGRAPH*, pages 48-1–48-7, 2007.

[2] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organisation of information. *Sov. Math. Dokl.*, 3:1259–1262, 1962.

[3] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Geometric approximation via coresets. In *Combinatorial and Computational Geometry, MSRI*, pages 1–30. University Press, 2005.

[4] I. Althofer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete Comput. Geom.*, 9(1):81–100, 1993.

[5] A. Amir, A. Efrat, P. Indyk, and H. Samet. Efficient regular data structures and algorithms for dilation, location, and proximity problems. *Algorithmica*, 30(2):164–187, 2001.

[6] A. Andersson. General balanced trees. *J. Algorithms*, 30:1–18, 1999.

[7] S. Arya, T. Malamatos, and D. M. Mount. A simple entropy-based algorithm for planar point location. *ACM Trans. Algorithms*, 3, 2007. (article: 17).

[8] S. Arya, T. Malamatos, and D. M. Mount. Space-time tradeoffs for approximate nearest neighbor searching. *J. Assoc. Comput. Mach.*, 57(1):1–54, 2009.

[9] S. Arya, T. Malamatos, D. M. Mount, and K.-C. Wong. Optimal expected-case planar point location. *SIAM J. Comput.*, 37:584–610, 2007.

[10] S. Arya and D. M. Mount. Algorithms for fast vector quantization. In *Data Compression Conference*, pages 381–390. IEEE Press, 1993.

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

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

[13] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *J. Assoc. Comput. Mach.*, 45:891–923, 1998.

[14] S. Arya, D. M. Mount, and M. Smid. Dynamic algorithms for geometric spanners of small diameter: Randomized solutions. *Comput. Geom. Theory Appl.*, 13:91–107, 1999.

[15] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[16] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):545–568, 1985.

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

[18] J. L. Bentley. Decomposable searching problem. *Inform. Process. Lett.*, 8:244–251, 1979.

[19] J. L. Bentley and J. B. Saxe. Decomposable searching problem I: Static-to-dynamic transformation. *J. Algorithms*, 1:301–358, 1980.

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

[21] S. N. Bespamyatnikh. Dynamic algorithms for approximate neighbor searching. In *Proc. Eighth Canad. Conf. Comput. Geom.*, pages 252–257, 1996.

[22] M. Bădoiu, R. Cole, E. D. Demaine, and J. Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theo. Comp. Sci.*, 382:86–96, 2007.

[23] P. B. Callahan. *Dealing with higher dimensions: The well-separated pair decomposition and its applications*. PhD dissertation, Johns Hopkins University, Department of Computer Science, 1995.

[24] P. B. Callahan and S. R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proc. Fourth Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 291–300, 1993.

[25] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. Assoc. Comput. Mach.*, 42:67–90, 1995.

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

[27] T. M. Chan. Closest-point problems simplified on the RAM. In *Proc. 13th Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 472–473, 2002.

[28] S. Chib and E. Greenberg. Understanding the Metropolis-Hastings algorithm. *The American Statistician*, 49:327–335, 1995.

[29] M. Cho, D. M. Mount, and E. Park. Maintaining nets and net trees under incremental motion. In *Proc. 20th Annu. Internat. Sympos. Algorithms Comput.*, pages 1134–1143, 2009.

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

[31] K. L. Clarkson. Approximation algorithms for shortest path motion planning. In *Proc. 19th Annu. ACM Sympos. Theory Comput.*, pages 56–65, 1987.

[32] R. Cole. On the dynamic finger conjecture for splay trees. Part 2: The proof. *SIAM J. Comput.*, 30:44–85, 2000.

[33] R. Cole and L.-A. Gottlieb. Searching dynamic point sets in spaces with bounded doubling dimension. In *Proc. 38th Annu. ACM Sympos. Theory Comput.*, pages 574–583, 2006.

[34] R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. Part 1: Splay sorting log n-block sequences. *SIAM J. Comput.*, 30:1–43, 2000.

[35] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 2005.

[36] A. Czumaj and A. Lingas. Fast approximation schemes for euclidean multi-connectivity problems. In *Proc. 27th Internat. Colloquium on Automata, Languages and Programming*, pages 856–868, 2000.

[37] S. P. Dandamudi and P. G. Sorenson. Algorithms for bd trees. *Softw. Pract. Exper.*, 16(12):1077–1096, 1986.

[38] M. de Berg, M. Roeloffzen, and B. Speckmann. Kinetic convex hulls and delaunay triangulations in the black-box model. In *Proc. 27th Annu. Sympos. Comput. Geom.*, pages 244–253, 2011.

[39] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, Germany, 3rd edition, 2008.

[40] E. D. Demaine, D. Harmon, J. Iacono, and M. Pătraşcu. The geometry of binary search tree. In *Proc. 20th Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 496–505, 2009.

[41] J. Derryberry, D. Sheehy, M. Woo, and D. D. Sleator. Achieving spatial adaptivity while finding approximate nearest neighbors. In *Proc. 20th Canad. Conf. Comput. Geom.*, pages 163–166, 2008.

[42] C. A. Duncan. *Balanced Aspect Ratio Trees.* PhD dissertation, Johns Hopkins University, Department of Computer Science, 1999.

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

[44] A. Elmasry. On the sequential access theorem and deque conjecture for splay trees. *Theo. Comp. Sci.*, 314:459–466, 2004.

[45] D. Eppstein, M. T. Goodrich, and J. Z. Sun. The skip quadtree: A simple dynamic data structure for multidimensional data. *Internat. J. Comput. Geom. Appl.*, 18:131–160, 2008.

[46] A. M. Farley, A. Proskurowski, D. Zappala, and K. J. Windisch. Spanners and message distribution in networks. *Discrete Applied Mathematics*, 137(2):159–171, 2004.

[47] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[48] J. Fischer and S. Har-Peled. Dynamic well-separated pair decomposition made easy. *Proc. 17th Canad. Conf. Comput. Geom.*, pages 235–238, 2005.

[49] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.*, 14(4):781–798, 1985.

[50] Greg N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24:37–65, 1997.

[51] I. Galperin and R. R. Rivest. Scapegoat trees. In *Proc. Fourth Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 165–174, 1993.

[52] J. Gao, L. Guibas, and A. Nguyen. Deformable spanners and applications. *Comput. Geom. Theory Appl.*, 35(1):2–19, 2006.

[53] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25:905–910, 1982.

[54] G. F. Georgakopoulos. Chain-splay trees, or, how to achieve and prove log logN-competitiveness by splaying. *Inform. Process. Lett.*, 106:37–43, 2008.

[55] L.-A. Gottlieb and L. Roditty. An optimal dynamic spanner for doubling metric spaces. In *Proc. 16th Annu. European Sympos. Algorithms*, volume 5193/2008, pages 478–489. Springer, 2008.

[56] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73:325–348, 1987.

[57] J. Gudmundsson, C. Levcopoulos, and G. Narasimhan. Improved greedy algorithms for constructing sparse geometric spanners. In *Proc. Seventh Scand. Workshop Algorithm Theory*, pages 314–327, 2000.

[58] J. Gudmundsson, G. Narasimhan, and M. Smid. Geometric spanners. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*, pages 360–364. Springer, 2008.

[59] L. Guibas. Kinetic data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and App.*, pages 23–1–23–18. Chapman and Hall/CRC, 2004.

[60] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19st Annu. IEEE Sympos. Found. Comput. Sci.*, pages 8–21, 1978.

[61] A. Gupta, R. Krauthgamer, and J. R. Lee. Bounded geometries, fractals, and low-distortion embeddings. In *Proc. 44th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 534–543, 2003.

[62] S. Har-Peled. A replacement for voronoi diagrams of near linear size. In *Proc. 42nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 94–103, 2001.

[63] S. Har-Peled. *Geometric Approximation Algorithms.* American Mathematical Society, Providence, Rhode Island, 2011.

[64] S. Har-Peled and M. Mendel. Fast construction of nets in low dimensional metrics and their applications. *SIAM J. Comput.*, 35:1148–1184, 2006.

[65] F. Harary. *Graph Theory.* Addison-Wesley, MA, 1969.

[66] P. D. Hoff, A. E. Raftery, and M. S Handcock. Latent space approaches to social network analysis. *J. American Statistical Assoc.*, 97:1090–1098, 2002.

[67] J. Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *Proc. 12th Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 516–522, 2001.

[68] J. Iacono. *Distribution-sensitive data structures.* PhD thesis, Rutgers, The state University of New Jersey, New Brunswick, New Jersey, 2001.

[69] J. Iacono. Expected asymptotically optimal planar point location. *Comput. Geom. Theory Appl.*, 29:19–22, 2004.

[70] J. Iacono. Key-independent optimality. *Algorithmica*, 42:3–10, 2005.

[71] J. M. Keil. Approximating the complete Euclidean graph. In *Proc. First Scand. Workshop Algorithm Theory*, pages 208–213, 1988.

[72] J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete euclidean graph. *Discrete Comput. Geom.*, 7(1):13–28, 1992.

[73] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[74] K. Knowlton. Progressive transmission of grey-scale and binary pictures by simple efficient, and lossless encoding schemes. *Proceedings of the IEEE*, 68:885–896, 1980.

[75] R. Krauthgamer and J. R. Lee. Navigating nets: Simple algorithms for proximity search. In *Proc. 15th Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 798–807, 2004.

[76] J. Matoušek. Geometric range searching. *ACM Comput. Surv.*, 26:421–461, 1994.

[77] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

[78] J. J. Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68(8):1703–1759, 2005.

[79] D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu. A computational framework for incremental motion. In *Proc. 20th Annu. Sympos. Comput. Geom.*, pages 200–209, 2004.

[80] D. M. Mount and E. Park. A dynamic data structure for approximate range searching. In *Proc. 26th Annu. Sympos. Comput. Geom.*, pages 247–256, 2010.

[81] G. Narasimhan and M. Smid. *Geometric Spanner Networks*. Cambridge University Press, 2007.

[82] G. Narasimhan and M. Zachariasen. Geometric minimum spanning trees via well-separated pair decompositions. *J. Exper. Algorithmics*, 6:1–27, 2001.

[83] G. Navarro, R. Paredes, and E. Chvez. t-spanners as a data structure for metric space searching. In *Proc. 9th Internat. Sympos. on String Processing and Information Retrieval*, pages 298–309, 2002.

[84] T. S. E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *Proc. 21st Annu. Joint Conf. of the IEEE Comp. and Commun. Societies*, pages 170–179, 2002.

[85] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM J. Comput.*, 2:33–43, 1973.

[86] Y. Ohsawa and M. Sakauchi. The bd-tree - a new n-dimensional data structure with highly efficient dynamic characteristics. In *IFIP Congress*, pages 539–544, 1983.

[87] E. Park and D. M. Mount. A self-adjusting data structure for multidimensional point sets. In *Proc. 20th Annu. European Sympos. Algorithms*, pages 778–789, 2012.

[88] E. Park and D. M. Mount. Output-sensitive well-separated pair decompositions for dynamic point sets. Unpublished manuscript, 2013.

[89] S. Pettie. Splay trees, Davenport-Schinzel sequences, and the deque conjecture. In *Proc. 19th Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 1115–1124, 2008.

[90] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33:668–676, 1990.

[91] S. Rao and W. D. Smith. Approximating geometrical graphs via spanners and banyans. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, pages 540–550, 1998.

[92] L. Roditty. Fully dynamic geometric spanners. *Algorithmica*, 62:1073–1087, 2012.

[93] J. Ruppert and R. Seidel. Approximating the d-dimensional complete Euclidean graph. In *Proc. Third Canad. Conf. Comput. Geom.*, pages 207–210, 1991.

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

[95] H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Trans. Pattern Anal. Mach. Intell.*, 10:579–586, 1988.

[96] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *IEEE Trans. on Knowledge and Data Engr.*, 22:1158–1175, 2010.

[97] R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

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

[99] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. Assoc. Comput. Mach.*, 32:652–686, 1985.

[100] M. Smid. Closest point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook on Computational Geometry*, pages 877–935. Elsevier Science, Amsterdam, 2000.

[101] M. Smid. The well-separated pair decomposition and its applications. In T. Gonzalez, editor, *Handbook of Approximation Algorithms and Metaheuristics*, pages 53-1–53-12. Chapman & Hall/CRC, Boca Raton, 2007.

[102] S. Solomon and M. Elkin. Balancing degree, diameter and weight in euclidean spanners. In *Proc. 18th Annu. European Sympos. Algorithms*, pages 48–59, 2010.

[103] A. Subramanian. An explanation of splaying. *J. Algorithms*, 20(3):512–525, 1996.

[104] R. Sundar. On the deque conjecture for the splay algorithm. *Combinatorica*, 12:95–124, 1992.

[105] K. Talwar. Bypassing the embedding: algorithms for low dimensional metrics. In *Proc. 36th Annu. ACM Sympos. Theory Comput.*, pages 281–290, 2004.

[106] R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5:367–378, 1985.

[107] J. B. Tenenbaum, B. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2323, 2000.

[108] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inform. Process. Lett.*, 40:175–179, 1991.

[109] P. M. Vaidya. An $o(n \log n)$ algorithm for the all-nearest-neighbors problem. *Inform. Process. Lett.*, 4(2):101–115, 1989.

[110] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. 16st Annu. IEEE Sympos. Found. Comput. Sci.*, pages 75–84, 1975.

[111] Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.

[112] C. C. Wang, J. Derryberry, and D. D. Sleator. $o(\log \log n)$-competitive dynamic binary search trees. In *Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 374–383, 2006.

[113] D. E. Willard. Polygon retrieval. *SIAM J. Comput.*, 11:149–165, 1982.

[114] K. Yi and Q. Zhang. Multi-dimensional online tracking. In *Proc. 20th Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 1098–1107, 2009.

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