Proximity Problems for Point Sets with Low Doubling Dimension

by

Lee-Ad Gottlieb

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

January 2009

Richard Cole—Advisor

Acknowledgements

Acknowledgement is first due to my advisor, Richard Cole. I thank him for patient direction, and devoting much of himself while expecting little in return. Now I find this medium inadequate to express my full gratitude, and must make do with a simple thank you.

My colleagues have made the past five years at Courant a very enjoyable period in my life. I am especially thankful to Shabsi Walfish and Raghavan Dhandapani for many conversations that were often useful and always amusing. I am also grateful to my friends (and co-authors) Tyler Neylon and Liam Roditty, who were always a pleasure to collaborate with.

Finally, I'd to thank my parents, Sigal and Lenny, and Zuki ("tech support") and Margalit for just being themselves.

Abstract

In this thesis we consider proximity problems on point sets. Proximity problems arise in many fields of computer science, with broad application to computational geometry, machine learning, computational biology, data mining, and the like. In particular, we consider the problems of approximate nearest neighbor search and dynamic maintenance of a spanner for a point set.

It has been conjectured that all algorithms for these two problems suffer from the "curse of dimensionality," which means that their run times grow exponentially with the dimension of the point set. To avoid this undesirable growth, we consider point sets that possess a doubling dimension λ. We first present a dynamic data structure that uses linear space and supports a $(1+\varepsilon)$-approximate nearest neighbor search of the point set. We then extend this data structure to allow the dynamic maintenance of a low degree $(1+\varepsilon)$-spanner for the point set. The query and update time of these structures are logarithmic in the size of the point set and exponential in λ (as opposed to exponential in the dimension); when λ is small, this provides a significant speed-up over known algorithms, and when λ and ε are constant these run times are optimal up to a constant. The spanner degree is optimal, while the spanner update times improve on all previously known algorithms.

Contents

Acknowledgements iv

Abstract v

1 Introduction **1**

1.1 Related work 5

1.1.1 Nearest neighbor search 5

1.1.2 Spanners 7

1.2 Thesis outline 8

2 A nearest neighbor search structure **10**

2.1 Hierarchical partition 11

2.2 Approximate nearest neighbor search 12

2.2.1 Tree extraction 12

2.2.2 Search description 13

2.3 Implicit representation of the hierarchy 17

2.4 Dynamic maintenance of the hierarchy 18

2.4.1 Insertions 18

2.4.2 Deletions 19

3 A new approximate nearest neighbor search structure 22

3.1 The hierarchy 23

3.2 Spanning tree 24

3.3 Implicit representation of the hierarchy 25

3.4 A cover search in $2^{O(\lambda)} \log n$ time 28

3.4.1 Balanced tree structure for T' 29

3.4.2 Search execution 30

3.4.3 Locating the lowest implicit covering point 34

3.5 Refinement search 35

3.6 Dynamic maintenance of the hierarchy 38

3.6.1 Insertions 38

3.6.2 Promotions 43

3.6.3 Correctness of the covering property 45

3.6.4 Deletions 49

3.7 Centroid Path Updates 52

3.7.1 Review of biased skip lists 53

3.7.2 Dynamic changes to T' 56

3.7.3 Modified biased skip lists 58

3.8 Linear space and efficient storage of friends lists 64

4 Application: A spanner 66

4.1 A first-attempt spanner 66

4.2 The new spanner 69
4.2.1 Motivation: An incremental spanner. 69
4.2.2 Step 1. Pruning the spanning tree 71
4.2.3 Step 2. Creating a better hierarchy 72
4.2.4 Step 3. A spanner for the intermediate hierarchy . . . 74
4.2.5 Step 4. A spanner for the final hierarchy 78
4.2.6 The degree of the final spanner 81
4.2.7 Dynamic updates 86

5 Further applications **93**

5.1 Closest pair 93
5.2 Well separated pairs decomposition 94

Chapter 1

Introduction

In 1961, Bellman [5] coined the now ubiquitous term "curse of dimensionality" to describe a problem that arises when estimating probability distribution functions (PDFs) on high dimensional spaces. The complexity of the solution, as well as the memory needed to compute it, grow exponentially with the dimension. The growing memory and computation requirements quickly cause the problem to become intractable.

Bellman himself referred only to estimating PDFs, but in fact the curse of dimensionality applies to a wide range of computational problems. For example, the curse can readily be seen when considering the convex hull of n points, perhaps one of the most basic features in computational geometry. Chazelle [15] gave an $O(n^{\lfloor d/2 \rfloor} + n \log n)$ algorithm for finding a convex hull (where n is the size of the point set and d is the dimension), and this algorithm is optimal. In fact, the maximum number of faces of a hull grows

exponentially in d; there may exist up to $n^{\lfloor d/2 \rfloor}$ faces of dimension $d-1$ [33]. It is not surprising then that other basic operations on the convex hull, such as determining the actual number of facets in the hull or whether the hull is simplicial (meaning the hull is composed solely of simplices of affine independent points), can be shown to require $\Omega(n^{\lfloor d/2 \rfloor - 1} + n \log n)$ time [20].

Now, consider the nearest neighbor search problem (sometimes called the post office problem). Nearest neighbor search (NNS) is one of the basic operations computed on points. The problem asks to preprocess a set X of points in a certain metric space M, so that given a new query point $q \in M$, the nearest point to q in X can be located efficiently.

In 1994, Clarkson [17] noted that although many data structures have been proposed for efficient NNS – kd-trees [7], for example, claimed an "empirically observed average running time of $O(\log n)$" – their query times generally feature a very steep dependence on d, at least $2^{\Omega(d)}$. Based on this, Clarkson concluded that the prospect of an $O(\log n)$ query time in high dimension is "crushed by the 'curse of dimensionality.' " While his claim remains unproven, there is partial evidence that points to an even stronger conjecture, that any (subexponential space) data structure for NNS will require exponential time for a query [10].

The aforementioned difficulties in NNS led researchers to consider an approximate version of this problem. The approximate nearest neighbor search problem (ANN) asks to preprocess a set X of points in a certain metric space M, so that given a new query point $q \in M$, *some point* near to

q in X can be located efficiently. This modified problem has applications in data mining, database queries and related fields.

However, for high dimensional metrics, a $(1+\varepsilon)$-approximate nearest neighbor search for $\varepsilon < 1$ may still require significant computation time, due to the inherent complexity of the metric: Currently the best query time for ANN in Euclidean space is $O(\varepsilon^{(1-d)/2} \log n)$ [14]. Hence it is natural to study ANN techniques for point sets which are effectively lower dimensional, although inhabiting a high dimensional space.

A recent successful approach has been to consider point sets that have a small *doubling dimension*: Let the space within radius r of a point be called the *ball* centered at that point. A point set X has doubling dimension λ if any set of points in X that are *covered* by a ball of radius r can be covered by 2^λ balls of radius $\frac{r}{2}$. A metric is *doubling* if its dimension is $O(1)$. While a low Euclidean dimension implies a low doubling dimension (Euclidean metrics of dimension d have doubling dimension $O(d)$ [24]), low doubling dimension is more general than low Euclidean dimension. For example, exact nearest neighbor in metric spaces with low doubling dimension may require $\Theta(n)$ computations (this follows easily from [25]), while for d-dimensional Euclidean space Clarkson [16] has given a size $O(n^{\lceil d/2 \rceil (1+\varepsilon)})$ data structure that answers exact nearest neighbor queries in $O(\log n)$ time (with constant factors in the bounds depending on d and ε, $\varepsilon > 0$). By contrast, for approximate nearest neighbor one can achieve the same results for low doubling dimension as are possible for low Euclidean dimension.

Krauthgamer and Lee [28] developed a search structure for approximate nearest neighbor search on a dynamic set of points. The performance of their algorithm depends on the doubling dimension. Let α be the *aspect ratio* of the points – the ratio between the largest and smallest interpoint distances. Then their data structure supports $(1+\varepsilon)$-approximate nearest neighbor searches in time $2^{O(\lambda)} \log \alpha + (1/\varepsilon)^{O(\lambda)}$, and allows updates in time $2^{O(\lambda)} \log \alpha$. However, the dependence on the aspect ratio in the algorithm of [28] is undesirable.

In this thesis we present a new data structure for approximate nearest neighbor searches on a dynamic set of points in a metric space that have doubling dimension λ. Our data structure has size $O(n)$ and supports insertions and deletions in $2^{O(\lambda)} \log n$ time, and finds a $(1+\varepsilon)$-approximate nearest neighbor in time $2^{O(\lambda)} \log n + (1/\varepsilon)^{O(\lambda)}$. These performance times are independent of the aspect ratio of the points.

Our data structure can be extended to allow the maintenance of a $(1+\varepsilon)$-spanner for S. A $(1+\varepsilon)$-spanner is a graph on the points S with the following property: Between any pair of points there is a path in the spanner whose total length is at most $(1+\varepsilon)$ times the actual distance between the points. We show how to construct a dynamic $(1+\varepsilon)$-spanner for a set of points in a metric space that have doubling dimension λ. The spanner has degree $\varepsilon^{-O(\lambda)}$ (which is optimal), and can be updated in $O(\frac{\log n}{\varepsilon^{O(\lambda)}})$ time.

Note that when λ and ε are constants, the NNS query time is $O(\log n)$, which matches the known lower bound on nearest neighbor search (in the

algebraic decision tree model [2]). Now, the task of inserting a point into a $(1+\varepsilon)$-spanner subsumes within it the task of discovering a $(1+\varepsilon)$-nearest neighbor of the new point, so it is optimal as well.

We first review related work (in Section 1.1), and then in Section 1.2 give an outline of the rest of the thesis.

1.1 Related work

We present an overview of related work on approximate nearest neighbor search and dynamic spanners.

1.1.1 Nearest neighbor search

Beygelzimer et al. [9] showed how to improve on the size of the data structure of Krauthgamer and Lee [28], requiring only $O(n)$ space for all operations except deletions. Krauthgamer and Lee [29] gave an alternate static data structure for ANN, obtaining results that are independent of the aspect ratio. Their static data structure uses space $O(n^3)$ and answers $(1+\varepsilon)$ queries in $2^{O(\lambda)} \log n + (1/\varepsilon)^{O(\lambda)}$ time.

Improving on this result, Har-Peled and Mendel [25] constructed a linear space ring–separator structure that provides an n^c-approximate nearest neighbor to the query point in $2^{O(\lambda)} \log n$ time (for constant c). While an n^c-approximation is very coarse, it provides enough information to "jump in" to a net-tree structure at a position that is within $O(\log n)$ levels of the

desired approximation (as discussed there). This clever observation completely circumvents the reliance on aspect ratio. However, the construction of [25] does not support insertions or deletions into the point set. Using the dynamization technique of Bentley and Saxe [8], this structure may be extended to allow for insertions, thereby producing a semi-dynamic structure, but then an insertion would require $2^{\Theta(\lambda)} \log^2 n$ time. It is not clear how to maintain the structure of [25] either dynamically or even semi-dynamically in $2^{O(\lambda)} \log n$ time given the special order that they impose on points when building their tree structure.

Other related research on nearest neighbor searches has focused on various assumptions concerning the metric space. Clarkson [18] made assumptions concerning the probability distribution from which X and q are drawn, and developed two randomized data structures for exact nearest neighbor. However, the query time is super-logarithmic, and the structures do not support insertion or deletion of points. Karger and Ruhl [26] introduced the notion of growth-constrained metrics (elsewhere called the KR-dimension) which is a weaker notion than that of the doubling dimension. They presented an $O(n \log n)$ space data structure on which a randomized algorithm finds an approximate nearest neighbor in $O(\log n)$ time, and which allows insertions and deletions in $O(\log n \log \log n)$ time. (These times hide a polynomial dependence on the expansion rate of the points.) A survey of proximity searches in metric space appeared in [13].

1.1.2 Spanners

A graph H is a t-spanner of G if $d_H(u,v) \leq t d_G(u,v)$, where $d_G(u,v)$ denotes the shortest path distance between u and v in G, and $d_H(u,v)$ denotes the shortest path distance between u and v in H. A spanner can also be defined for a set of points residing in Euclidean space: Let S be a set of points in $\Re^d$. The graph G is a complete graph whose vertices are the points of S, and the weight of every edge is the distance between its endpoints. A geometric t-spanner is then constructed on the graph G.

Geometric spanners have received a fair amount of attention in the past couple of decades. Various papers have dealt with the construction of geometric spanners with specific properties, such as linear number of edges, small weight (the weight of a spanner is the sum of the weights of its edges), small hop diameter and low degree. For points residing in (low) d-dimensional Euclidean space, Vaidya [32], Salowe [30], Callahan and Kosaraju [12] and Soares [31] showed how to compute a geometric $(1+\varepsilon)$-spanner with $O(n/\varepsilon^d)$ edges in $O(n \log n + \varepsilon^{-d} \log(1/\varepsilon)n)$ time. In the dynamic setting, where the problem is to *explicitly* maintain a set of edges that constitute a spanner of the point set, Arya *et al.* [3] obtained $O(\log^d n \log\log n)$ update time in the restricted model in which updates were assumed to be random: A point to be deleted is assumed to be selected at random from S, and a point to be inserted is assumed to be a random point of the new point set. Bose *et al.* [11] gave a semi-dynamic algorithm that supports insertions in $O(\log^{d-1} n)$ time. Gao *et al.* [22] considered both dynamic and kinetic spanners (a kinetic spanner

supports movement of the points), and gave a spanner with update time and degree $O(\frac{\log \alpha}{\varepsilon^d})$, where α is the *aspect ratio* of the set, the ratio between the largest and smallest interpoint distances of the set. This result is of interest when α is small, which is often the case.

We are interested in the question of dynamic spanners for points that reside in a metric space and have doubling dimension λ. Since doubling dimension is more general than Euclidean dimension, all results for doubling dimension apply to Euclidean dimension as well. In this setting Har-Peled and Mendel [25] showed how to construct a static constant degree spanner in $O(\frac{n \log n}{\varepsilon^{O(\lambda)}})$ time. Roditty [27] gave a dynamic spanner that supports insertions in $O(\frac{\log n}{\varepsilon^{O(\lambda)}})$ amortized time and deletions in $\widetilde{O}(\frac{n^{1/3}}{\varepsilon^{O(\lambda)}})$ amortized time (where the notation $\widetilde{O}$ is used to hide logarithmic factors). Very recently, Gottlieb and Roditty gave a dynamic spanner that supports insertions in $O(\frac{\log^2 n}{\varepsilon^{O(\lambda)}})$ amortized time and deletions in $O(\frac{\log^3 n}{\varepsilon^{O(\lambda)}})$ amortized time [23].

1.2 Thesis outline

In Chapter 2, we describe the hierarchy and nearest neighbor search structure of Krauthgamer and Lee [28]. In Chapter 3, we detail the new hierarchy and nearest neighbor search structure. In Chapter 4, we describe how to use the new hierarchy to maintain a better spanner. And in Chapter 5, we describe some further applications of the nearest neighbor search structure.

As a preliminary point, we note that it can be shown (via a repeated

application of the doubling property) that if set S has minimum inter-point distance a, then at most $(\frac{b}{a})^{O(\lambda)}$ points of S can be found within distance b of any $x \in S$: To see this, note that the points of S are covered by a ball of radius b, and so by the doubling dimension the points of S may be covered by 2^{λ} balls of size $\frac{b}{2}$, $2^{2\lambda}$ balls of size $\frac{b}{4}$, and $2^{O(\lambda \log(b/a))} = (\frac{b}{a})^{O(\lambda)}$ balls of size a. Since the minimum inter-point distance in S is a, each ball of size a can hold at most a single point, so we have an upper bound of $(\frac{b}{a})^{O(\lambda)}$ on the number of points in S.

Chapter 2

A nearest neighbor search structure

In what follows, we describe in detail the hierarchy introduced by Krauthgamer and Lee [28], and show how it may be used to support a $(1+\varepsilon)$-nearest neighbor search (NNS) in $2^{O(\lambda)} \log \alpha + \varepsilon^{-O(\lambda)}$ time. We also review how to update the hierarchy in $2^{O(\lambda)} \log \alpha$ time under insertions and deletions. ([28] claims a slightly worse bound for insertions and deletions, but actually $2^{O(\lambda)} \log \alpha$ suffices.) In its original presentation, the hierarchy required $2^{O(\lambda)} n$ space. In our presentation, we will incorporate a modification suggested by Beygelzimer *et al.* [9], coupled with a new technique, to bring the space requirement down to $O(n)$. (For ease of presentation, some layout and terminology introduced here may differ from the original presentation.)

2.1 Hierarchical partition

For a set Y, $X \subset Y$ is an r-discrete center set (or r-net) of Y if it satisfies the following two properties:

(i) Packing: For every pair $x, y \in X$, $d(x, y) \geq r$.

(ii) Covering: Every point $y \in Y$ is strictly within distance r of some point $x \in X$: $d(y, x) < r$.

We say that $x \in X$ *covers* $y \in Y$ if $d(x, y) < r$. The previous conditions require that the points of X be spaced out, yet cover all points of Y.

Let S be a set of points with doubling dimension λ and aspect ratio α. (For ease of presentation, we assume the minimum inter-point distance in S is 1, and that α is a power of 2 and is known in advance.) The hierarchical partition is a hierarchy of $\log \alpha + 1$ discrete center sets, $Y_1 = Y_{2^0}, Y_{2^1}, Y_{2^2}, \ldots, Y_\alpha$. The first (or bottom) level of the hierarchy is the set $Y_1 = S$. The i^{th} level, for $i > 0$, is the set Y_{2^i}, where Y_{2^i} a 2^i-discrete center set of $Y_{2^{i-1}}$. 2^i is the *radius* of set Y_{2^i}. Note that the final (or top) level of the hierarchy, Y_α, must contain a single point.

Let x^j denote the occurrence of point x in level Y_{2^j} of the hierarchy. x may appear in as many as $\log \alpha + 1$ levels of the hierarchy, which implies that the size of the hierarchy may be $\Theta(n \log \alpha)$. Later (in Section 2.3), we will show how an implicit representation of the hierarchy can be stored in $O(n)$ space.

2.2 Approximate nearest neighbor search

The hierarchy can be used to support a $(1+\varepsilon)$-NNS algorithm. To facilitate the search, we will first need to extract a spanning tree T that directly corresponds to the hierarchy.

2.2.1 Tree extraction

The extraction of the spanning tree for the hierarchy is straightforward. The nodes of T are arranged in $\log \alpha + 1$ levels, $T_0, \ldots, T_{\log \alpha}$, where the nodes in T_i store points of Y_{2^i}. For each point occurrence x_j^m, there exists a unique tree node $v_j \in T_m$ (or v_j^m) which stores x_j^m. Tree level T_0 – the leaf level – contains n leaf nodes, each one storing a unique point in S. Tree level $T_{\log \alpha}$ contains a single node, which is the root of the tree.

The tree edges are determined as follows. Consider each point x_j^m in turn, and let x_i^{m+1} be the point that covers x_j^m. (If more than one point covers x_j^m, let x_i^{m+1} be the closest one of these.) Node v_j^m is connected to v_i^{m+1}, and is its tree child.

The edge assignments define an ancestral relationship among the nodes of T. We will extend this ancestral relationship to the corresponding points of the hierarchy: If v_i^l is a parent (or ancestor) of v_j^m $(l > m)$, then we say that x_i^l is a parent (ancestor) of x_j^m. Note that a node or point may have at most $2^{O(\lambda)}$ children.

The following property of tree nodes and hierarchical points follows by

construction:

Property 1. *Let x_i^l be an ancestor of x_j^m, or equivalently, let v_i^l be an ancestor of v_j^m, $l > m$. Then $d(x_i^l, x_j^m) < \sum_{k=m+1}^{l} 2^k = 2 \cdot 2^l - 2^{m+1}$.*

2.2.2 Search description

Now that we have extracted the spanning tree T from the hierarchy, we can describe the $(1+\varepsilon)$-NNS algorithm. The $(1+\varepsilon)$-nearest neighbor search consists of two separate stages: The first stage is a *4-cover search* which requires $2^{O(\lambda)} \log \alpha$ work. The second stage is a *refinement search*, which requires $\varepsilon^{-O(\lambda)}$ work. The total run time of the $(1+\varepsilon)$-NNS is $2^{O(\lambda)} \log \alpha + \varepsilon^{-O(\lambda)}$.

4-cover search.

A point y^l in the hierarchy *c-covers* a point q, if $d(y^l, q) < c \cdot 2^l$. If y^l c-covers q, and q is not c-covered by any point in all levels below Y_{2^l}, then Y_{2^l} is the lowest level in which q is c-covered. When c is a constant, there may be $2^{\Theta(\lambda)}$ points in level Y_{2^l} that c-cover q.

The 4-cover search on q uses the spanning tree to identify all points that 4-cover q. The key observation driving the search is that if z^m 4-covers q, then all ancestors of z^m 4-cover q as well. To see this, first note that $d(z^m, q) < 4 \cdot 2^m$, and then recall that by Property 1 the distance from z^m to any ancestor y^l is less than $2 \cdot 2^l - 2^{m+1}$. It follows that the distance

from q to y^l is $d(y^l, q) \leq d(y^l, z^m) + d(z^m, q) < (2 \cdot 2^l - 2^{m+1}) + 4 \cdot 2^m = 2 \cdot 2^l - 2^{m+1} + 2 \cdot 2^{m+1} = 2 \cdot 2^l + 2^{m+1} \leq 2 \cdot 2^l + 2^l = 3 \cdot 2^l$. Hence, y^l 4-covers q. (Note that in fact this same property is true of 2-covering. The need for a four cover search will become clear in the proof of Lemma 2.2.1 below.)

The 4-cover search on T begins with the root node at level $T_{\log \alpha}$. If the point stored at this node does not 4-cover q, then no point 4-covers q, and the algorithm terminates. If the stored point does 4-cover q, then the root node is placed in set $V_{\log \alpha}$. At each iteration, the search descends down one level of the tree. At level T_m, the search inspects the children of nodes in V_{m+1}; if any of these children store points that 4-cover q, they are placed in set V_m. The search terminates at level T_0, or earlier if it encounters a level in which no nodes store points that 4-cover q.

The search finds all points that 4-cover q; correctness follows from the aforementioned observation that if a point 4-covers q, all its ancestors 4-cover q as well. A set V_m may contain at most $2^{O(\lambda)}$ nodes, from which it follows that the search considers $2^{O(\lambda)}$ nodes at each of the $O(\log \alpha)$ levels, and terminates in $2^{O(\lambda)} \log \alpha$ time.

Refinement search

The refinement search follows the completion of the 4-containment search. Recall that the 4-cover search identifies the lowest level T_m that contains nodes which store points that 4-cover q. V_m is the set of these nodes.

The refinement search backtracks one step, taking the set V_{m+1}. This set

consists of nodes in level T_{m+1} that 4-cover q. The refinement search proceeds to identify the set V' comprising *all* level $T_{m-\log(4/\varepsilon)}$ descendants of the nodes in V_{m+1}. This can be done by descending $\log(4/\varepsilon)+1$ further levels in T. When ε is appropriately small, the cost of this search is bound by the size of V', which is $\varepsilon^{-O(\lambda)}$. After determining V', we inspect the points stored in the nodes of V' and identify the point which is closest to q. We can prove the following:

Lemma 2.2.1. *Let z be the nearest neighbor of q in S.*

(i) $d(q,z) > 2\cdot 2^{m-1}+1$.

(ii) At least one node in V' stores a point $x^{m-\log(4/\varepsilon)}$ that satisfies $d(q,x^{m-\log(4/\varepsilon)}) < \frac{1}{2}2^m\varepsilon - 1 + d(z,q)$.

(iii) $x^{m-\log(4/\varepsilon)}$ is a $(1+\varepsilon)$-approximate nearest neighbor of q.

Proof. Recall that z^0 is the occurrence of z in the bottom level of the hierarchy.

(i) A lower bound on $d(z,q) = d(z^0,q)$ follows. q is not 4-covered by any points in level $Y_{2^{m-1}}$, so the distance from q to all these points is at least $4\cdot 2^{m-1}$. By Property 1, the distance from q to all descendants of these points at the bottom level of the hierarchy is greater than $4\cdot 2^{m-1} - (2\cdot 2^{m-1}-1) = 2\cdot 2^{m-1}+1$. Since z^0 is a descendant of some point in $Y_{2^{m-1}}$, $d(z^0,q) > 2\cdot 2^{m-1}+1$.

(ii) Correctness follows in two steps. The first step establishes that an ancestor of z^0 is stored by some node in V'; we name this ancestor $x^{m-\log(4/\varepsilon)}$. The second step establishes that $d(q, x^{m-\log(4/\varepsilon)}) < \frac{1}{2}2^m\varepsilon - 1 + d(z, q)$.

The first step establishes that $x^{m-\log(4/\varepsilon)}$, the ancestor of z^0 in level $Y_{2^{m-\log(4/\varepsilon)}}$, is stored by some node in V'. To prove this, it suffices to show that some node in V_{m+1} stores an ancestor of z^0 (which is of course also the ancestor of $x^{m-\log(4/\varepsilon)}$). Since V' includes all tree descendants of this node at level $T_{m-\log(4/\varepsilon)}$, V' must contain a node storing $x^{m-\log(4/\varepsilon)}$.

Since q is 4-covered by some points in Y_{2^m}, the distance from q to these points is less than $4 \cdot 2^m$. Since z is the nearest neighbor of q, it must be that $d(z^0, q) < 4 \cdot 2^m$ as well. By Property 1, the distance from z^0 to its ancestor in level Y_{m+1}, say w^{m+1}, is less than $2 \cdot 2^{m+1} - 1$. It follows that $d(w^{m+1}, q) \leq d(w^{m+1}, z^0) + d(z^0, q) < (2 \cdot 2^{m+1} - 1) + 4 \cdot 2^m = 2 \cdot 2^{m+1} - 1 + 2 \cdot 2^{m+1} = 4 \cdot 2^{m+1} - 1$. It follows that w^{m+1} 4-covers q, and so w^{m+1} must be stored by some node in V_{m+1}. As mentioned above, a consequence of this is that $x^{m-\log(4/\varepsilon)}$ must be stored by some node in V'.

The second step establishes an upper bound on $d(x^{m-\log(4/\varepsilon)}, q)$. By Property 1, $d(x^{m-\log(4/\varepsilon)}, z^0) < 2 \cdot 2^{m-\log(4/\varepsilon)} - 1 = \frac{1}{2} \cdot 2^m\varepsilon - 1$. It follows that $d(x^{m-\log(4/\varepsilon)}, q) \leq d(x^{m-\log(4/\varepsilon)}, z^0) + d(z^0, q) < \frac{1}{2} \cdot 2^m\varepsilon - 1 + d(z^0, q)$.

(iii) It follows from (i) and (ii) that $x^{m-\log(4/\varepsilon)}$ is a $(1+\delta)$-approximate nearest neighbor of q, with $1 + \delta = \frac{d(x^{m-\log(4/\varepsilon)}, q)}{d(z^0, q)} < \frac{\frac{1}{2} \cdot 2^m\varepsilon - 1 + d(z^0, q)}{d(z^0, q)} = 1 + \frac{\frac{1}{2} \cdot 2^m\varepsilon - 1}{d(z^0, q)} < 1 + \frac{\frac{1}{2} \cdot 2^m\varepsilon - 1}{2 \cdot 2^{m-1} + 1} < 1 + \frac{\frac{1}{2} \cdot 2^m\varepsilon}{2 \cdot 2^{m-1}} = 1 + \frac{\varepsilon}{2} < 1 + \varepsilon$. □

It follows that some node of V' stores a $(1+\varepsilon)$-approximate nearest neighbor of q. We search all stored point in the nodes of V' and identify the point closest to q. This point must also be a $(1+\varepsilon)$-approximate nearest neighbor of q. We conclude that the cover and refinement searches find a $(1+\varepsilon)$-approximate nearest neighbor in $2^{O(\lambda)} \log \alpha + \varepsilon^{-O(\lambda)}$ steps.

2.3 Implicit representation of the hierarchy

Here we review how the hierarchy and associated tree may be stored implicitly in $O(n)$ space. This section incorporates ideas from [28, 9].

A single-child path is a maximal chain of parent-child nodes where each node (including the final one) has only a single child. Compress all single-child paths in T, retaining only the first node on the path, and connect this first node to the single child of the final path node; these are now a parent-child pair. This parent-child pair of nodes in distant levels implies the existence of the removed intermediate nodes, which are thereby represented implicitly. Each internal node in the resulting tree either has minimum degree 2, or has a child with minimum degree two. Since the tree has exactly n leaves (one storing each point), the size of the tree is $O(n)$. We remove from the hierarchy all points that were stored in the removed nodes, and the size of the hierarchy is the same as the size of the tree.

The cover and refinement searches proceed in a top-down fashion as before. When a search encounters a compressed path, it fills in the implicit

nodes necessary for the search.

2.4 Dynamic maintenance of the hierarchy

In this section, we review how to support dynamic updates to the hierarchy and its spanning tree in $2^{O(\lambda)} \log \alpha$ time.

2.4.1 Insertions

The insertion of a point z into an empty hierarchy is accomplished by inserting an instance of z into all $\log \alpha + 1$ levels of the hierarchy.

To insert a point z into a nonempty hierarchy, we must first find the lowest point y^m that 1-covers (or simply covers) z. (If there are multiple points in level Y_{2^m} that cover z, we take the closest one.) y^m can be found in $2^{O(\lambda)} \log \alpha$ time by executing a 4-cover search while keeping track of all points that 1-cover z. Since all points that 1-cover z necessarily 4-cover z as well, these points will be encountered by the 4-cover search.

Once y^m is identified, we insert z^{m-1} as its child. (If y^m was previously represented implicitly in the hierarchy, it now has two children and will appear explicitly.) We further insert z into every level of the hierarchy below level $Y_{2^{m-1}}$, where occurrence z^p is the child of z^{p+1} ($m-1 > p \geq 0$); however, the compression scheme implies that only z^{m-1} and z^0 will appear explicitly in the hierarchy. It is clear that these points obey the covering property, and since z is not covered by any points in levels $Y_{2^{m-1}}$ and lower, these points

obey the packing property as well.

T is updated with a single node storing each explicit point occurrence. The insertion run-time is dominated by the 4-cover search, and is $2^{O(\lambda)} \log \alpha$.

2.4.2 Deletions

We present a deletion scheme that requires $2^{O(\lambda)} \log \alpha$ time. The deletion schemes presented in [28, 9] needed a data structure using $2^{\Theta(\lambda)} n$ space, while the following scheme employs a data structure using $O(n)$ space.

When a point z is deleted from the point set, all occurrences of z in the hierarchy (and the tree nodes storing these occurrences) are marked as deleted, but no further action is taken. This scheme results in two issues that need to be addressed: (i) After multiple deletions, there may be n' nodes in the tree but only n points in the point set, and (ii) the 4-cover and containment searches may return a node that has no real (non-deleted) descendants.

The first concern is addressed by rebuilding the hierarchy and tree in the background. It suffices, if T holds n' nodes (included those nodes storing deleted points), to start rebuilding after $\frac{n'}{3}$ deletions, and to complete the rebuilding over the next $\frac{n'}{6}$ insertions and deletions; that is, for each update to the point set 7 updates are performed on the background structure. The completed hierarchy (and tree) will then contain at least $\frac{n'}{2}$ points, including at most $\frac{n'}{6}$ deleted points.

To address the second concern, we record which nodes in the tree have at

least one leaf descendant storing a non-deleted point. This can be done by, after each update, traversing the tree from the leaf level up to the root level (in $O(\log \alpha)$ time), and recording at each internal node which of its children have a leaf descendant storing a non-deleted point.

The procedure for insertions or deletions (including the 4-cover search) proceeds as before, without regard to which nodes have leaf descendants storing non-deleted points. However, when searching for an approximate nearest neighbor the cover and refinement searches ignore a node if none of its leaf descendants store a non-deleted point.

Modifications to the refinement search

Recall that the refinement search identified a set of nodes V', inspected the points stored in this set, and returned the closest one. Under our deletion scheme, when a node $v \in V'$ stores a deleted point, the refinement scheme will consider in its stead an arbitrary non-deleted point stored in some leaf descendant of v (if there are any such points). We can show that this does not affect the correctness of the refinement search, which still returns a $(1+\varepsilon)$-approximate nearest neighbor. Revisiting Lemma 2.2.1, we take $x^{m-\log 4/\varepsilon}$ as the deleted point stored in v, and say that the refinement search considered in its stead a descendant $\tilde{z}^0$. By Property 1 we have that $d(x^{m-\log 4/\varepsilon}, \tilde{z}^0) < 2^{m-\log(4/\varepsilon)} - 1 = \frac{1}{2}2^m\varepsilon - 1$. It follows that $d(q, \tilde{z}^0) \leq d(q, x^{m-\log(4/\varepsilon)}) + d(x^{m-\log(4/\varepsilon)}, \tilde{z}^0) < (\frac{1}{2}2^m\varepsilon - 1 + d(q,z)) + (\frac{1}{2}2^m\varepsilon - 1) = 2^m\varepsilon - 1 + d(q,z)$. $\tilde{z}$ is therefore a $(1+\delta)$-approximate nearest neighbor of q, where $1+\delta = \frac{d(q,\tilde{z}^0)}{d(q,z^0)} <$

$$\frac{2^m\varepsilon-1+d(q,z^0)}{d(q,z^0)} < 1+\frac{2^m\varepsilon-1}{d(q,z^0)} < 1+\frac{2^m\varepsilon-1}{2\cdot 2^{m-1}+1} < 1+\frac{2^m\varepsilon}{2\cdot 2^{m-1}} = 1+\varepsilon.$$

Chapter 3

A new approximate nearest neighbor search structure

The hierarchy of [28] supports approximate nearest neighbor searches in $2^{O(\lambda)} \log \alpha + \varepsilon^{-O(\lambda)}$ time and updates in $2^{O(\lambda)} \log \alpha$ time. These times are acceptable when $\alpha = n^{O(1)}$ – which is often the case – but problematic when α is asymptotically larger. Note that the introduction of α into the nearest neighbor search and update times is due solely to the 4-cover search, which requires $2^{O(\lambda)} \log \alpha$ time.

Here we show how to construct and maintain a modified hierarchy that supports a cover search in $2^{O(\lambda)} \log n$ time. This improved run time eliminates the dependence on α in the nearest neighbor search and update times. The description of the hierarchy is straightforward, although the cover search is more involved and handling dynamic updates is intricate. As before, many

points of the hierarchy will be represented implicitly.

A spanning tree is maintained for the hierarchy. A balanced tree structure for the spanning tree drives the cover search. The final update and search times are reduced to $2^{O(\lambda)} \log n$ and $2^{O(\lambda)} \log n + \varepsilon^{O(\lambda)}$, respectively.

3.1 The hierarchy

Let us first present the new hierarchy. (For ease of presentation, we assume that the minimum possible inter-point distance in S is 1. The hierarchy can be maintained without this assumption, and without knowledge of the minimum inter-point distance in advance.) The hierarchy contains an infinite number of levels. The first (or bottom) level of the hierarchy is the set $Y_1 = Y_{5^0} = S$, and the top level $Y_{5^\infty = \infty}$ contains only a single point. Each intermediate level $0 < i < \infty$ is represented by a set Y_{5^i} which is a 5^i-discrete center set of $Y_{5^{i-1}}$, where the definition of a r-discrete center set is slightly altered to satisfy the following properties:

(i) Packing: For every pair $x, y \in X$, $d(x, y) \geq \frac{1}{5}r$.

(ii) Covering: Every point $y \in Y$ is strictly within distance $\frac{3}{5}r$ of some point $x \in X$: $d(x, y) < \frac{3}{5}r$.

The radius of level Y_{5^m} is defined to be 5^m. We have altered the *dropoff* of the hierarchy – the ratio of radii from one level to the next – to 5 instead of 2; the purpose for this will become clear in Section 3.6.2. A point y^l covers

a point x if $d(y^l, x) < 5^l$, and the covering property implies that all points in the hierarchy are $\frac{3}{5}$-covered.

We further stipulate that for each point occurrence x^m, the hierarchy records all points within distance $2 \cdot 5^m$ of x^m. These points are the *friends* of x^m, and are recorded in the *friends list* for x^m. Friends list are necessary for the execution of the cover search. $x_{j,i}$ may have at most $2^{O(\lambda)}$ friends. (The astute reader may have noted that the storage of all the friends lists may require $2^{\Theta(\lambda)}n = \omega(n)$ space. We address this issue in Section 3.8, where we give a space saving technique that reduces the storage requirement to $O(n)$.)

3.2 Spanning tree

The extraction of the spanning tree for the hierarchy proceeds as before. The nodes of T are arranged in an infinite number of levels $T_{-\infty}, \ldots, T_{\infty}$, where each node of level T_m stores a unique point of Y_{5^m}. Let the node storing point x_j^m be v_j^m. The tree edges are determined in the usual way: For a point x_j^m, let x_i^{m+1} be the point that $\frac{3}{5}$-covers x_j^m. (If multiple points $\frac{3}{5}$-cover x_j^m, a single one of these is chosen based on the insertion and promotion rules below.) v_j^m is connected to v_i^{m+1}, and is its tree child.

Two nodes are friends if the points stored in them are friends. We extend the ancestral relationship of the tree to apply to the corresponding points of the hierarchy, precisely as was done before. This gives us the following

close-covering property that relates ancestors and descendants.

Property 2. Close-covering*: Let x_i^l be an ancestor of x_j^m, or equivalently let let v_i^l be an ancestor of v_j^m. Then* $d(x_i^l, x_j^m) \leq \sum_{k=m+1}^{l} \frac{3}{5}5^k = \frac{3}{4}5^l - \frac{3}{4}5^m = \frac{4}{5}5^l - 5^m - \frac{1}{20}5^l + \frac{1}{4}5^m \leq \frac{4}{5}5^l - 5^m$.

3.3 Implicit representation of the hierarchy

Although each point of the hierarchy appears in an infinite number of levels, there exists a simple implicit representation of the hierarchy which stores only $O(n)$ points. As we show below, the implicit representation does not interfere with the cover or refinement searches, so we can still find the lowest point in the hierarchy that covers q, even if this point is represented implicitly.

We present here the tree compression scheme for T, which implies an implicit representation of the hierarchy. (We present a general overview of the scheme, with more specifics and some exceptions deferred to Sections 3.6.1 and 3.6.2.) As before, we compress all single-child paths in T, with only the first node remaining uncompressed. The first node is connected to the single child of the last node, and such a parent-child pair of nodes in non-adjacent levels implies the existence of the intermediate nodes. However, there is a caveat to this compression: A node v on a single-child path remains uncompressed if the point stored by v has friends in its level. (As mentioned above, the execution of the cover search will use the friends lists, and as a result points with friends need to be represented explicitly.) Let the resulting

tree be T'. The compression scheme implies an implicit representation of the hierarchy.

As described above, the new implicit representation differs from the old one only in that points with friends are explicitly stored. The following lemma shows that there are only $O(n)$ point occurrences which have friends, so the size of the tree and the explicit hierarchy remains $O(n)$.

Lemma 3.3.1. *The number of point occurrences in the hierarchy having friends is $O(n)$.*

Proof. The bound follows from a simple charging argument. Consider a point z, and let Y_{5^l} be the highest level in which z is present in the hierarchy. The three occurrences z^l, z^{l-1} and z^0 are changed to z if they have friends. Any other occurrences of z that has friends is charged to an arbitrary one of its friends. We will prove that z can be charged at most twice for occurrences of other points, from which it follows that z can be charged at most five times.

Let z be charged for some occurrence of y – say z^m and y^m are friends. By assumption, y^m is not the highest or second highest occurrence of y in the hierarchy (or else y^m would be charged to y and not to z), so y exists in level $Y_{5^{m+2}}$. We can show that (i) z^m must be the highest or second highest occurrence of z in the hierarchy; and (ii) z^m has no other friends that can be charged to z. These two facts together imply that z can only be charged twice for occurrences of other points.

We first show that z^m must be the highest or second highest occurrence of z in the hierarchy. Suppose in contradiction that z existed in level $Y_{5^{m+2}}$ of the

hierarchy. Then we can show z^{m+2} violates the packing property with respect to y^{m+2}: Since y^m and z^m are friends, $d(y,z) \leq 2 \cdot 5^m = \frac{2}{25}5^{m+2} < \frac{1}{5}5^{m+2}$, which implies that y^{m+2} and z^{m+2} violate the packing property of level $Y_{5^{m+2}}$. It follows that z cannot exist in level $Y_{5^{m+2}}$, and z^m is the highest or second highest occurrence of z.

Now we show that z^m has no other friends in level Y_{5^m} that can be charged to z. Suppose in contradiction that z^m has a friend x^m that is charged to z. By assumption, x^m is not the highest or second highest occurrence of x (or else x^m would be charged to x), so x exists in level $Y_{5^{m+2}}$ of the hierarchy. We show that x^{m+2} violates the packing property with respect to y^{m+2}: Recalling that both x^m and y^m are friends of z^m, we have that $d(x,y) \leq d(x,z) + d(z,y) \leq 2 \cdot 5^m + 2 \cdot 5^m = 4 \cdot 5^m = \frac{4}{25}5^{m+2} < \frac{1}{5}5^{m+2}$, which implies that x^{m+2} and y^{m+2} violated the packing property of level $Y_{5^{m+2}}$. It follows that there does not exist a second point x^m charged to z.

We conclude that each point z can be charged at most five times for points with friends, and since every point occurrence is charged to that point or to its friend, there are $O(n)$ points in the hierarchy with friends. □

When a point is inserted into the hierarchy, it may cause $2^{O(\lambda)}$ friends which were previously represented implicitly to now appear explicitly. Nevertheless, Lemma 3.3.1 demonstrates that in total $O(n)$ points appear explicitly due to possessing friends. The insertion scheme will be presented in Sections 3.6.1 and 3.6.2, where we will see that each insertion causes $O(1)$ additional points to be stored explicitly (in addition to the $2^{O(\lambda)}$ points which

are represented explicitly due to their having friends). We may conclude that the total number of explicit points in the hierarchy is $O(n)$.

3.4 A cover search in $2^{O(\lambda)} \log n$ time

While a 4-cover search was necessary to support nearest neighbor searches and updates for the previous hierarchy, the construction of the new hierarchy is such that a 1-cover (or just 'cover') search suffices. A cover search on q finds the lowest level points which cover q. Given these points, we can execute a refinement search to find a $(1+\varepsilon)$-nearest neighbor of q in $(1/\varepsilon)^{O(\lambda)}$ more steps. The following lemma is the key observation motivating the cover search.

Lemma 3.4.1. *(i) If z^m covers q, then all ancestors of z^m cover q.*

(ii) If both z^m and x^l $(l > m)$ cover q, then z^m is a descendant of x^l or of one of x^l's friends.

Proof. **(i)** By the close-covering property, the distance from z^m to its ancestor y^l is not greater than $\frac{4}{5}(5^l - 5^m)$. We have that $d(y^l, q) \leq d(y^l, z^m) + d(z^m, q) < (\frac{4}{5}5^l - 5^m) + 5^m = \frac{4}{5}5^l < 5^l$, so y^l covers q.

(ii) It follows from (i) that y^l, the ancestor of z^m, covers q. Since x^l also covers q, we have that $d(x^l, y^l) \leq d(x^l, q) + d(y^l, q) \leq 5^l + 5^l = 2 \cdot 5^l$, from which it follows that if x^l and y^l are not the same point, then they are friends. □

(Note that in fact this same property is true of a $\frac{4}{5}$-covering. The need for a 1-cover search will become clear in the proof of Lemma 3.5.1.)

The cover search presented below first finds the lowest *explicitly* represented point in the hierarchy that covers q (or the set of such points if there is more than one). This point may not be the lowest point covering q, since that point may be represented *implicitly*. However, we will show (in Section 3.4.3) that given the lowest explicitly represented point that covers q, it is an easy matter to identify the lowest point that covers q. This will complete the description of the cover search.

The execution of the cover search makes use of the compressed spanning tree T' of the hierarchy and also the friends list of each point. To attain the $2^{O(\lambda)} \log n$ search time, we will need to maintain a balanced tree structure for T', and it will drive the search.

3.4.1 Balanced tree structure for T'

To support the $2^{O(\lambda)} \log n$ time cover search, we maintain a *centroid path* decomposition of T', and for each centroid path store its nodes in a weighted search structure. To this end, define $s(v)$, the size of node v, to be the number of nodes in the subtree rooted at v.

For our purposes, we define the centroid path of a tree to be the path starting at the root, which at each node v branches to v's *largest* child (the child with the greatest size), with ties broken arbitrarily. In addition, we stipulate that each centroid path C_i contains nodes of size $2^j < s(v) \leq 2^{j+1}$

for appropriate i; j is the *scale* of C_i. In a centroid path decomposition, we recursively decompose each off-path subtree of the centroid path. Note that in the centroid path decomposition the largest scale is $j = \lceil \log n \rceil$, and also that a path from the root to a leaf traverses at most $\log n$ distinct centroid paths.

Let $w(v)$, the *weight* of a node v, be the number of nodes in its off-path subtrees, plus one for the node itself. The nodes of each centroid path, ordered top-down, are stored in a weighted search structure. We will use the biased skip lists of Bagchi *et al.* [4].

The maintenance of the centroid path decomposition and associated biased skip lists is intricate, and is deferred to Section 3.7.

We can now describe the cover search.

3.4.2 Search execution

In this section, we show how to locate the lowest explicitly maintained point that covers the query point q.

Suppose for the moment that the points of the hierarchy were well-separated, meaning that the distance between all points in level Y_{5^m} is greater then $2 \cdot 5^m$, and so no point has any friends. Then it would be an easy matter to execute an $O(\log n)$ time cover search for q using T'. The search begins at the root of T', and is defined throughout by v, the current node of interest which stores a point that covers q, and by the centroid path C_i that contains v. By the well-separated assumption, at most one point at each level cov-

ers q. As a consequence of Lemma 3.4.1, the search need only consider the descendants of this one point.

The search is divided into phases. In each phase, the search considers the current centroid path C_i, and the task is to find v', the bottommost node on C_i which stores a point that covers q. The search first checks if the bottom node on C_i, say v_b, stores a point that covers q. If it does then $v' = v_b$. If v_b's point does not cover q, then v' is located by means of a weighted binary search driven by the weighted search structure for the centroid path C_i. At each step, the binary search tests consecutive nodes $v_1, v_2 \in C_i$, where v_1 is the parent of v_2; if v_1's stored point covers q but v_2's point does not, then $v' = v_1$. Otherwise the search continues in the appropriate part of C_i (everything strictly above v_1, or everything strictly below).

Having found v', the search tests each off-path child of v' to see if any of them stores a point that covers q. If so, that child is the new node of interest, and the search proceeds to the next phase with the new node of interest and its centroid path. (Note that the scale of the new centroid path is less than the scale of C_i, so the number of search phases is $O(\log n)$.) If not, the search terminates at v'.

This algorithm runs in $2^{\lambda} \log n$ time. In general, each step of the weighted search for v' on C_i removes a constant fraction of the weight of C_i from consideration. Once v' is found, the search inspects the children of v' ($2^{O(\lambda)}$ time) to find the new node of interest and proceeds with the child and its centroid path. This implies that the entire search can be executed in $2^{O(\lambda)} \log n$ time.

However, we must also consider the case where v' is the bottom node of C_i; in this case $w(v')$ may be similar to W, and the weighted search does not remove a constant fraction of the weight of C_i from consideration. But in this event the search identifies v' in a single step, finds the child that is the new node of interest in $2^{O(\lambda)}$ time, and begins the next phase. Recalling that the search consists of only $O(\log n)$ phases we conclude that the search time remains bounded by $2^{O(\lambda)} \log n$.

The difficulty in using this search procedure is that the points of the hierarchy need not be well-separated. If we find that q is not covered by the point stored in v then we may indeed eliminate v from contention, since Lemma 3.4.1 ensures that none of the descendants of v stores a point covering q; but if we find that q is covered by v's point, we cannot eliminate v's friends from consideration. This problem would appear to break the logarithmic search, but can be evaded by noting that the search scheme of [28] succeeded by considering $2^{O(\lambda)}$ points at each level. Translated into our context, this implies that only $2^{O(\lambda)}$ subtrees (or centroid paths) need be retained at each step.

More formally, let a search be defined by a set V of nodes of interest. V is a set of nodes in the same tree level that store points that cover q. As a consequence of Lemma 3.4.1, the search need only consider the descendants of these points. (Another consequence of the Lemma is that these points must be friends.) For each node $v_i \in V$, we record the centroid path C_i of T in which v_i is found, or rather the *portion* of C_i that has not yet been

searched. Let C be the set of these partial centroid paths.

The search begins with the root as the only member of V. At every step, the search takes the partial centroid path C_i whose weight is largest, and as in the ideal well-separated case tests the bottom node of C_i (if not already tested). If the bottom node's stored point does not cover q, the search tests consecutive nodes of C_i, and as before prunes a constant fraction of the weight of C_i. If neither of these nodes store points that cover q, then the search continues by considering the current heaviest partial path.

When, for some i, a node v_i is found whose stored point covers q, the search from v_i continues as follows. All children of v_i are tested to determine if their stored points cover q. Let v_i' be a child of v_i containing q, if any. Then v_i' and its friends that store points which cover q form the new set of nodes of interest. (Note that by Lemma 3.4.1, the lowest points covering q must be descendants of these points.) The search then continues with V consisting of v_i' and these friends; for each one of these nodes, we record its full centroid path C_j, or the surviving portion of C_j if C_j has already been pruned. If v_i has no child v_i' that stores a point that covers q, then the new set of nodes of interest include v_i (with its partial path C_i consisting of just v_i) and those friends of v_i that store points that cover q (with their partial centroid paths determined as before). If this set is empty, the search stops at v_i. The search terminates when all partial paths are empty, or have been reduced to single nodes that have no children storing points that cover q.

Lemma 3.4.1 implies that the new set of the nodes of interest are descen-

dants of nodes in the old set of nodes of interest. It follows that the new nodes of interest are either found on the partial paths in C, or are found on paths that are descendants of the partial paths in C, and a descendant path must necessarily be of lesser weight than its ancestral path. More precisely, let the weight of the heaviest partial path C_i be W, consider the set of partial paths $C' \subset C$ with weight greater than $\frac{W}{2}$, and recall that $|C'| \leq |C| = 2^{O(\lambda)}$. A search step replaces C_i by one or more paths with weight not greater than $\frac{W}{2}$. Each other path of C' may be replaced by some other paths as well, but by at most one path of weight greater than $\frac{W}{2}$. It follows that a single search step reduces the size of $|C'|$ by at least one, and that $2^{O(\lambda)}$ steps reduce $|C'|$ to 0. It follows that:

Theorem 3.4.2. *The above search procedure on T terminates in $2^{O(\lambda)} \log n$ time.*

3.4.3 Locating the lowest implicit covering point

The cover search described above returns the lowest explicitly represented point covering q (or the set of such points). If the lowest point covering q (say z^p) is indeed represented explicitly, then it will be returned by the cover search. If z^p is not represented explicitly, then the cover search will not locate it.

To circumvent this difficulty, we prove that the cover search must in fact return some occurrence of z, say z^m $(m > p)$. That is, we show that there

exists an occurrence z^m which is the lowest explicit point covering q. Given z^m and inspecting its distance from the query point, it is trivial to deduce that z^p (which by construction must exist) also covers q.

We now show that the cover search must return some occurrence of z. First consider all occurrence of z above z^p, and note that at least one of them (the highest one, say z^l) must be represented explicitly, since its parent has at least two children. Further note that by Lemma 3.4.1, each one of these occurrences covers q. Let Y_{5^m} ($l \geq m$) be the lowest level in which q is covered by some explicit point y^m. By Lemma 3.4.1(ii), y^m and z^m are friends so z^m must be found explicitly in the hierarchy. z^m is therefore returned by the cover search.

3.5 Refinement search

In the last section, we showed how to execute a cover search to find the lowest points in the hierarchy that covers q. The refinement search uses the lowest containing points to find a $(1+\varepsilon)$ nearest neighbor of q, as was done above in Section 2.2.2.

Let Y_{5^m} be the lowest level containing points that cover q. The cover search returns a set containing all points in Y_{5^m} that cover q. (If there is only one point in Y_{5^m} that covers q, then it is possible that this point is represented only implicitly and has no storing node. The refinement search will fill in this node, and all other implicit nodes encountered from here on.) We backtrack

one step and take the set V_{m+1} of all nodes in level T_{m+1} that cover q. We then identify the set V' consisting of all level $T_{5^{m-\log_5 \frac{40}{\varepsilon}}}$ descendants of the nodes in V_{m+1}. This may be accomplished by descending $\log \frac{40}{\varepsilon} + 1$ further levels in the tree. The cost of this search is bounded by the size of V', which is $(1/\varepsilon)^{-O(\lambda)}$. After determining V', we inspect the points stored in the nodes of V' and identify the point which is closest to q. We can prove the following:

Lemma 3.5.1. *Let z be the nearest neighbor of q in S.*

(i) $d(q, z) > \frac{1}{5} \cdot 5^{m-1} + 1$.

(ii) At least one node in V' stores a point $x^{m-\log_5(40/\varepsilon)}$ that satisfies $d(q, x^{m-\log(4/\varepsilon)}) < \frac{1}{50}5^m\varepsilon - 1 + d(z, q)$.

(iii) $x^{m-\log_5(40/\varepsilon)}$ is a $(1+\varepsilon)$-approximate nearest neighbor of q.

Proof. Recall that z^0 is the occurrence of z in the bottom level of the hierarchy.

(i) A lower bound on $d(z^0, q)$ follows. q is not covered by any points in level $Y_{5^{m-1}}$, so the distance from q to all these points is at least 5^{m-1}. By Property 2, the distance from q to all descendants of these points at the bottom level of the hierarchy is greater than $5^{m-1} - (\frac{4}{5}5^{m-1} - 1) = \frac{1}{5}5^{m-1} + 1$. Since z^0 is a descendant of some point in $Y_{5^{m-1}}$, $d(z^0, q) > \frac{1}{5}5^{m-1} + 1$.

(ii) Correctness follows in two steps. The first step establishes that an ancestor of z^0 is stored by some node in V'; we name this ancestor $x^{m-\log_5(40/\varepsilon)}$. The second step establishes that $d(q, x^{m-\log(4/\varepsilon)}) < \frac{1}{50}5^m\varepsilon - 1 + d(z, q)$.

The first step establishes that $x^{m-\log_5(40/\varepsilon)}$, the ancestor of z^0 in level $Y_{5^{m-\log_5(40/\varepsilon)}}$, is stored by some node in V'. To prove this, it suffices to show that some node in V_{m+1} stores an ancestor of z^0 (which is of course also the ancestor of $x^{m-\log_5(40/\varepsilon)}$). Since V' includes all tree descendants of this node in level $Y_{5^{m-\log_5(40/\varepsilon)}}$, V' must contain a node storing $x^{m-\log_5(40/\varepsilon)}$.

Since q is covered by some points in Y_{5^m}, the distance from q to these points is less than 5^m. Since z^0 is the nearest neighbor of q, it must be that $d(z^0, q) < 5^m$ as well. By Property 2, the distance from z^0 to its ancestor in level Y_{m+1}, w^{m+1} say, is less than $\frac{4}{5} \cdot 5^{m+1}$. It follows that $d(w^{m+1}, q) \leq d(w^{m+1}, z^0) + d(z^0, q) < \frac{4}{5} \cdot 5^{m+1} + 5^m = 5^{m+1}$. It follows that w^{m+1} covers q, and so w^{m+1} must be stored by some node in V_{m+1}. As mentioned above, a consequence of this is that $x^{m-\log_5(40/\varepsilon)}$ must be stored by some node in V'.

The second step establishes an upper bound on $d(x^{m-\log_5(40/\varepsilon)}, q)$. By Property 2, $d(x^{m-\log_5(40/\varepsilon)}, z^0) < \frac{4}{5} 5^{m-\log_5(40/\varepsilon)} - 1 = \frac{1}{50} \cdot 5^m \varepsilon - 1$. It follows that $d(x^{m-\log_5(40/\varepsilon)}, q) \leq d(x^{m-\log_5(40/\varepsilon)}, z^0) + d(z^0, q) = \frac{1}{50} \cdot 5^m \varepsilon - 1 + d(z^0, q)$.

(iii) It follows from (i) and (ii) that $x^{m-\log_5(40/\varepsilon)}$ is a $(1+\delta)$-approximate nearest neighbor of q, with $1 + \delta = \frac{d(x^{m-\log_5(40/\varepsilon)}, q)}{d(z^0, q)} = \frac{\frac{1}{50} \cdot 5^m \varepsilon - 1 + d(z^0, q)}{d(z^0, q)} = 1 + \frac{\frac{1}{50} \cdot 5^m \varepsilon - 1}{d(z^0, q)} < 1 + \frac{\frac{1}{50} \cdot 5^m \varepsilon - 1}{\frac{1}{5} 5^{m-1} + 1} < 1 + \frac{\frac{1}{50} \cdot 5^m \varepsilon}{\frac{1}{5} 5^{m-1}} = 1 + \frac{1}{2}\varepsilon < 1 + \varepsilon$. □

It follows that some node of V' stores a $(1+\varepsilon)$-approximate nearest neighbor of q. We search all stored points in the nodes of V' and identify the point closest to q. This point must also be a $(1+\varepsilon)$-approximate nearest neighbor of q. We conclude that the cover and refinement searches find a

$(1+\varepsilon)$-approximate nearest neighbor in $2^{O(\lambda)} \log n + \varepsilon^{-O(\lambda)}$ steps.

3.6 Dynamic maintenance of the hierarchy

In this section, we discuss how to maintain the hierarchy under dynamic updates in $2^{O(\lambda)} \log n$ time. An insertion into the hierarchy is similar to an insertion into the hierarchy of [28] (with some additional complications), and the runtime is dominated by the cost of a cover search for the new point. An insertion requires adding to the hierarchy at most $2^{O(\lambda)}$ explicit points that have friends, and $O(1)$ points that lack friends. (The size of the explicit hierarchy will remain $O(n)$). We will see that maintaining the covering condition of the hierarchy in the desired time bound requires some new ideas.

Maintaining the hierarchy under deletions will be done by simply marking the leaf nodes as deleted. This will necessitate rebuilding in the background. We will show that, as before, the nearest neighbor search can avoid nodes that have no descendants storing non-deleted points.

3.6.1 Insertions

When inserting a point, we wish to follow roughly the same general approach that was used in updating the hierarchy of Krauthgamer and Lee [28]. We could, for example, suggest a 'naive' scheme in which a cover search identifies the lowest point that covers the inserted point z, say y^m, and insert z^{m-1}

as its child. We would then insert z into every level of the hierarchy below $Y_{5^{m-1}}$, where occurrence z^p is the child of z^{p+1} $(m-1 > p \geq 0)$. (Only z^{m-1} and z^0 would appear explicitly). However, such an insertion scheme still leaves us with two obstacles to overcome.

The first obstacle is that z must appear explicitly in every level in which it has friends, and it is conceivable that there are many such levels. However, we can show that the close-covering property ensures that of all occurrences of z, only z^{m-1} and z^{m-2} could have friends: Recall that z is not covered by any point in level $Y_{5^{m-1}}$, so the distance from z to any point in this level is at least 5^{m-1}. By Property 2, the distance from z to any descendant of these points is greater than $5^{m-1} - \frac{4}{5}5^{m-1} = \frac{1}{5}5^{m-1} = 5 \cdot 5^{m-3} > 2 \cdot 5^{m-3}$. It follows that occurrences of z at levels $Y_{5^{m-3}}$ or lower have no friends. (In fact, this observation is the major motivation behind Property 2, the close-covering property.)

The second obstacle is that the insertion technique above does not maintain the covering invariant of the hierarchy (and by extension, the close-covering property). This is because z^{m-1} is not necessarily strictly within distance $\frac{3}{5}5^m$ of its covering point y^m. We could perhaps suggest an alternate insertion scheme, where we search for the lowest point that $\frac{3}{5}$-covers z, and then insert z beginning at the level below this point; but this level could be far above level Y_{5^m}, and it can be shown that all occurrences of z above level Y_{5^m} have friends and therefore must be represented explicitly. Hence, this suggested scheme may result in too much work per insertion. Instead, we

will retain the insertion scheme above, modify it only slightly, and then show what additional steps are necessary to guarantee that the covering property is maintained. Let us first describe the precise insertion rules we will use.

Insertion Rule 1. *(IR1) The insertion of z into an empty hierarchy proceeds as follows: We insert z into all levels of the hierarchy $Y_1, \ldots, Y_\infty$.*

Each occurrence z^p is a child of z^{p+1} (except for the top occurrence). Only the top and bottom occurrences of z are stored explicitly. After the first insertion, all points obey the packing and covering properties.

Insertion Rule 2. *(IR2) The insertion of z into a non-empty hierarchy proceeds as follows: A cover search is executed to locate the lowest level point that covers z. Let this point be y^m. (If there are multiple such points, let y^m be the closest one.)*

(i) z is inserted in all levels from $Y_{5^{m-1}}$ down.

(ii) If $d(z, y^m) > \frac{1}{5}5^m$, then z^m is inserted as well.

With the exception of the top occurrence of z, each occurrences z^p is a child of z^{p+1}. If z^{m-1} is the top occurrence of z then it is a child of y^m. All points inserted by IR2(i) obey the covering property (in fact, they are $\frac{1}{5}$-covered), and since z is not covered by any points in level $Y_{5^{m-1}}$ or lower, all occurrences obey the packing property as well. Now consider the case when IR2(ii) inserts z^m as the top occurrence of z. By assumption, z^m obeys the packing property. We would like to assign z^m as a child of the closest point

in $Y_{5^{m+1}}$; however, we have not guaranteed that this point $\frac{3}{5}$-covers z^m and satisfies the covering property. Guaranteeing the existence of such a point is the outstanding issue in the insertion scheme, and the goal of the promotion scheme of the next section.

The top and bottom occurrences of z are stored explicitly. As we have shown above, z^{m-1} and z^{m-2} may have friends and be explicitly stored, but no other occurrences of z have friends, so only these are stored explicitly. The insertion of z^m, z^{m-1} and z^{m-2} also causes friends of these points to be stored explicitly. So an insertion adds $O(1)$ explicit points that do not have friends and $2^{O(\lambda)}$ explicit points that have friends.

(For simplicity, once a point z^m is assigned a parent y^{m+1}, it is never be assigned a new parent x^{m+1}, even if $d(z^m, x^{m+1}) < d(z^m, y^{m+1})$. Giving a point a new parent would require a cut and link operation on the balanced tree structure and is too expensive.)

In closing this section, we note that this insertion scheme closely resembles the naive insertion scheme, with the only change being the additional point added by rule IR2(ii). The insertion of this additional point is in fact a critical feature that will enable us to guarantee the covering property. To see why IR2(ii) is useful, let us consider a point y^m that has no children other than y^{m-1}. In the naive insertion scheme the next point insertion (say of point z_1) may add to y^m a child z_1^{m-1} at distance $\frac{3}{5}5^m$ or greater, immediately violating the covering property. In the new insertion scheme, the insertion of z_1 can only add to y^m a child z_1^{m-1} at distance less than $\frac{1}{5}5^m$: If z_1 is distance

$\frac{1}{5}5^m$ or greater from y^m, the insertion of z_1^{m-1} would trigger IR(ii) and insert z_1^m as a parent of z_1^{m-1}. Hence, a single insertion cannot cause the covering property to be violated.

However, a covering violation below y^m can result from a sequence of four insertions:

- The first insertion is a point z_1 at distance almost 5^{m-1} from y^{m-1}; IR2(i) inserts z_1^{m-2} and IR2(ii) inserts its parent z_1^{m-1} (covered by y^m).
- The second insertion is z_2 at distance almost 5^{m-1} from z_1^{m-1} and almost $2 \cdot 5^{m-1}$ from y^m; IR2(i) inserts z_2^{m-2} and IR2(ii) inserts its parent z_2^{m-1} (covered by y^m).
- The third insertion is z_3 at distance almost 5^{m-1} from z_2^{m-1} and almost $3 \cdot 5^{m-1}$ from y^m; IR2(i) inserts z_3^{m-2} and IR2(ii) inserts its parent z_3^{m-1} (covered by y^m).
- The fourth insertion is z_4 at distance almost 5^{m-1} from z_3^{m-1} and more than $3 \cdot 5^{m-1} = \frac{3}{5}5^m$ from y^m; IR2(i) inserts z_3^{m-2} and IR2(ii) inserts its parent z_3^{m-1}. z_3^{m-1} is not covered by y^m.

In effect, IR2(ii) has allowed us to 'buy time,' deferring by several insertions the time at which the covering property is violated. This deferral will provide time for *promotions*, which we describe in the next section. The promotion scheme will guarantee that the covering property is maintained.

3.6.2 Promotions

The insertion rules ensure that several points must be inserted as the children of a point y^m before one of them will be at distance $\frac{3}{5}5^m$ or greater from y^m and violate the covering property. The multiple insertions over which this occurs gives us an opportunity to *promote* one of these children. (A promotion of a point occurrence creates an occurrence of the point one level up.) The point created by the promotion will $\frac{3}{5}$-cover points that are too far away from y^m, and will serve as their parent. However, such a promotion scheme could result in a difficulty, if the point created by the promotion is itself not properly covered. Hence, a subtler approach is called for.

Obligations. When a point is inserted into the hierarchy by IR(ii) or by a promotion, it may be given an *obligation*; an obligation is the name of some point which is a candidate for promotion at a later time. Let z^m be an inserted point, and let y^{m+1} be the closest point in level $Y_{5^{m+1}}$. The obligation rules are determined by $d(z^m, y^{m+1})$ as follows.

(i) If $\frac{1}{5}5^{m+1} \leq d(z^m, y^{m+1}) < \frac{2}{5}5^{m+1}$, then the obligation of z^m is the point that is the obligation of y^{m+1}.

(ii) If $\frac{2}{5}5^{m+1} \leq d(z^m, y^{m+1}) < \frac{4}{5}5^{m+1}$, then the obligation of z^m is itself.

(Recall that points inserted by IR2(i) are always $\frac{1}{5}$-covered when they are inserted; hence, they cannot have an obligation. This is why only points inserted by IR2(ii) or by a promotion can have an obligation.)

Triggering promotions. A promotion is triggered by the insertion of z^m in the vicinity of y^{m+1}, the closest point in the next highest level. The promotion rules are determined by $d(z^m, y^{m+1})$ as follows:

(i) If $\frac{2}{5}5^{m+1} \leq d(z^m, y^{m+1}) < \frac{3}{5}5^{m+1}$, then y^{m+1}'s obligation is promoted.

(ii) If $\frac{3}{5}5^{m+1} \leq d(z^m, y^{m+1}) < \frac{4}{5}5^{m+1}$, then z^m's obligation (which is itself) is promoted, resulting in the addition of z^{m+1} to the hierarchy. z^{m+1} is the parent of z^m.

The new point occurrence created by the promotion is made a child of the closest point at the next highest level. We will show that this new occurrence is $\frac{3}{5}$-covered by the closest point at the next highest level, and that the above obligation and promotion rules suffice to guarantee that the covering property is maintained.

We will also show (in Lemma 3.6.1 below) that a point created by a promotion cannot itself trigger another promotion. Since points inserted by IR(i) are $\frac{1}{5}$-covered, they cannot trigger promotions either. It follows that only an application of IR(ii) can trigger a promotion. We may conclude that a point insertion accounts for the addition of a single promoted point into the hierarchy; this point is be stored explicitly (even if it has no friends). A promotion may further cause $2^{O(\lambda)}$ friends of the promoted points to be stored explicitly.

To ensure correctness of the packing property, we stipulate that before a point x^p is promoted, we check whether x^p is $\frac{1}{5}$-covered by some point w^{p+1}.

(This is only possible if w^{p+1} was added subsequent to the creation of x^p. It suffices to examine the friends of x^p's parent, if any, at level $Y_{5^{p+1}}$.) If x^p is $\frac{1}{5}$-covered by w^{p+1}, then the promotion of x^p would violate the packing property, and so the promotion is abandoned. We will see that this extra stipulation does not cause a difficulty in the promotion scheme, since w^{p+1} will take the place of x^{p+1}.

3.6.3 Correctness of the covering property

We now prove that the above insertion and promotion rules preserve the covering property. In proving this, it will be helpful to introduce some terminology. (Note that this terminology only alludes to concepts introduced by the obligation and promotion rules, and does not introduce new concepts.) Let z^m be a point, and let y^{m+1} be the closest point to z^m in level $Y_{5^{m+1}}$.

Definition 1. *z^m is supersafe if*

(i) *z^∞ is the root; or*

(ii) *$d(z^m, y^{m+1}) < \frac{1}{5}5^{m+1}$; or*

(iii) *$\frac{1}{5}5^{m+1} \le d(z^m, y^{m+1}) < \frac{2}{5}5^{m+1}$, and y^{m+1} is supersafe.*

z^m is safe if

(iv) *$\frac{1}{5}5^{m+1} \le d(z^m, y^{m+1}) < \frac{2}{5}5^{m+1}$, and y^{m+1} is safe; or*

(v) *$\frac{2}{5}5^{m+1} \le d(z^m, y^{m+1}) < \frac{3}{5}5^{m+1}$, and y^{m+1} is supersafe.*

Less formally, if a *newly added* point z^m is supersafe, then it is properly covered and has no obligation. If the newly added point is safe, it is properly covered and has an obligation. It is however possible for z^m to become supersafe at a later time. This can occur, for example, if z^m is promoted, or if a newly added point covers z^m in a way that makes z^m supersafe. Then z^m will have an obligation even though it is supersafe.

It is important to note that it is possible to have an entire chain of parent-child points which all match definition **(iv)**, and hence are all safe. All these points will have the same obligation, which is the point at the top of the chain. If any one of these safe points subsequently becomes supersafe (either by the promotion of the obligation point that causes the obligation to become supersafe, or by the addition of a new point that covers one of the safe points in a way that causes it to become supersafe), then all its descendants in the parent-child chain will now match definition **(iii)**, and will all be supersafe.

The next lemma proves that all points are safe or supersafe, implying that all points are properly covered, and that the covering property is maintained. It further shows that a promotion cannot trigger a second promotion, which implies that only a single promotion may follow a point insertion.

Lemma 3.6.1. *(i) All points created by an insertion are safe or supersafe.*

(ii) All points created by a promotion are safe or supersafe. Further, a promotion does not trigger a second promotion.

Proof. **(i)** Points created by IR1 and IR2(i) are $\frac{1}{5}$-covered, so they are su-

persafe. We turn to points created by IR2(ii). Let z^m be inserted by IR2(ii), and let x^{m+1} be the closest point to z^m in level $Y_{5^{m+1}}$. The following cases are possible:

(i) $d(z^m, x^{m+1}) < \frac{1}{5}5^{m+1}$. Then z^m is supersafe.

(ii) $\frac{1}{5}5^{m+1} \leq d(z^m, x^{m+1}) < \frac{2}{5}5^{m+1}$. z^m is supersafe if x^{m+1} is supersafe, and safe if x^{m+1} is safe.

(iii) $\frac{2}{5}5^{m+1} \leq d(z^m, x^{m+1}) < \frac{3}{5}5^{m+1}$. z^m is safe if x^{m+1} is supersafe. If x^{m+1} is safe, then z^m provokes a promotion of the point that is x^{m+1}'s obligation at a higher level. This promotion is caused by an obligation that is common to an entire chain of safe points from the promoted point down to x^{m+1}. As mentioned above, the promotion causes the entire chain to become supersafe, and so z^m is now safe.

(iv) $\frac{3}{5}5^{m+1} \leq d(z^m, x^{m+1}) < \frac{4}{5}5^{m+1}$. z^m is promoted and the existence of z^{m+1} implies that z^m is supersafe.

(v) $\frac{4}{5}5^{m+1} \leq d(z^m, x^{m+1}) < 5^{m+1}$. We show that this situation is not possible. For z^m to be inserted by IR2(ii) at this level, there must be a point y^m which covers z, $d(y^m, z) < 5^m$. By assumption, y^{m+1} does not exist in the hierarchy, since x^{m+1} is the closest point to z in this level. It follows that y^m was not promoted, and so the distance from y^m to its parent w^{m+1} is less than $\frac{3}{5}5^{m+1}$. But then $d(w^{m+1}, z^m) \leq$

$d(w^{m+1}, y^m) + d(y^m, z^m) < \frac{3}{5}5^{m+1} + 5^m < \frac{4}{5}5^{m+1}$, which contradicts the assumption that x^{m+1} is the closest point to z in level $Y_{5^{m+1}}$.

(ii) Assume that the Lemma held prior to this promotion. Let z^m be a point that is promoted to create z^{m+1}, and further assume that the Lemma held prior to this promotion. We will first show that there exists a point y^{m+1} that covers z^m in level $Y_{5^{m+1}}$ and is supersafe.

The first case is when z^m was itself created by a promotion. Let y^{m+1} be the point in $Y_{5^{m+1}}$ closest to z^m. By assumption, z^m is $\frac{3}{5}$-covered by y^{m+1}; $d(z^m, y^{m+1}) < \frac{3}{5}5^m$. Now, the promotion of z^m implies that z^m had itself as an obligation; this allows us to place a lower bound on $d(z^m, y^{m+1})$, so that $\frac{2}{5}5^{m+1} \leq d(z^m, y^{m+1}) < \frac{3}{5}5^m$. Now, if y^{m+1} had had an obligation, the addition of z^m at this distance would have triggered the obligation. But by assumption a promotion does not trigger another promotion, so y^{m+1} has no obligation, and it is supersafe.

The second case is when z^m was created directly by an insertion. Since z^m has itself as an obligation, there exists a point y^{m+1} which satisfies $\frac{2}{5}5^{m+1} \leq d(z^m, y^{m+1}) < \frac{4}{5}5^m$. The insertion of z^m at this distance implies that y^{m+1} must have a child $\tilde{z}^m$ for which $\frac{2}{5}5^{m+1} \leq d(\tilde{z}^m, y^{m+1}) < \frac{3}{5}5^{m+1}$. ($\tilde{z}$ is either a child inserted before z, or $\tilde{z} = z$.) Then y^{m+1} must be supersafe, or else $\tilde{z}^m$ could not even be safe.

We have proved that y^{m+1} is supersafe. Let x^{m+2} be the $Y_{5^{m+2}}$ point closest to y^{m+1}. If y^{m+1} is supersafe because $d(y^{m+1}, x^{m+2}) < \frac{1}{5}5^{m+2}$, then $d(z^{m+1}, x^{m+2}) \leq d(z, y) + d(y, x) < \frac{4}{5}5^{m+1} + \frac{1}{5}5^{m+2} < \frac{2}{5}5^{m+2}$; in this case

z^{m+1} does not trigger a promotion, and also z^{m+1} is safe or supersafe. If y^{m+1} is supersafe because $d(y^{m+1}, x^{m+2}) < \frac{2}{5}5^{m+2}$, and x^{m+2} is supersafe, then $d(z^{m+1}, x^{m+2}) \leq d(z, y) + d(y, x) < \frac{4}{5}5^{m+1} + \frac{2}{5}5^{m+2} < \frac{3}{5}5^{m+2}$; in this case z^{m+1} does not trigger a promotion (since x^{m+2} is supersafe and has no obligation), and also z^{m+1} is safe. □

3.6.4 Deletions

When a point z is deleted, z^0 is marked as deleted, but no other changes are made to the hierarchy or to the spanning tree T'. This scheme results in two issues that need to be addressed: (i) After multiple deletions, there may be b points in the hierarchy but $o(b)$ points in the set, and (ii) the cover search for nearest neighbor search may return a point that is not the ancestor of a real point. The first concern is addressed by rebuilding the data structure in the background. (It suffices, if the hierarchy stores n points including deleted points, to start rebuilding when it includes $\frac{n}{3}$ deletions, and to complete the rebuilding over the next $\frac{n}{6}$ insertions and deletions; i.e. for each update, to perform 7 updates on the background structure. Once rebuilt, the hierarchy will contain at least $\frac{n}{2}$ points including at most $\frac{n}{6}$ deleted points.)

To address the second problem, we devise a scheme for returning only points which are ancestors of real points. For each centroid path, we keep track of whether its top node has any leaf descendants that store non-deleted points; if so, we also record if the bottommost node of the centroid path has leaf descendants that store non-deleted points, and we record which (if any)

of the path's nodes have leaf descendants storing non-deleted points in their off-path subtrees. Call these nodes *real nodes*. For each centroid path, the real nodes are kept in a standard balanced tree. We organize this tree so that its leftmost node stores the bottommost real node of the centroid path.

Note that if a path loses some (but not all) of its leaf descendants that store non-deleted points, then each ancestral path continues to have the same real nodes, and therefore its balanced tree remains unchanged. Thus, the effect of deleting a point is to reduce a (possibly empty) series of paths from having one real node to having none, and for the next path up to decrease its number of real nodes to some nonzero number. All paths further up are unaffected. The cost of the updates to the associated balanced trees is $O(1)$ for each of $O(\log n)$ paths in the series, and $O(\log n)$ for the topmost path, for a total of $O(\log n)$.

The insertion scheme proceeds irrespective of which nodes have descendants storing non-deleted nodes, so a containment search for insertions proceeds as before, irrespective of which nodes are real. For a containment search for an approximate nearest neighbor search however, we wish to return only nodes that have descendants storing real nodes. To this end, when the search algorithm considers a centroid path, we seek a node on this path that has leaf descendants storing non-deleted points in addition to containing the query point. When the nearest neighbor search begins the refinement search, we again only consider nodes that have leaf descendants storing non-deleted points. We can determine in $O(1)$ time whether a node has leaf descendants

storing non-deleted points, by locating its centroid path and the lowest node of that path that contains real descendants; this descendant is the leftmost node of the balanced tree for real nodes, so it can be identified in $O(1)$ time.

Modifications to the refinement search.

Recall that the refinement search identified a set of nodes V', inspected the points stored in this set, and returns the closest one. Under our deletion scheme, when a node $v \in V'$ stores a deleted point, the refinement scheme will consider in its stead an arbitrary non-deleted point stored in some leaf descendant of v (if there are any such points). We can show that this does not affect the correctness of the refinement search, which still returns a $(1+\varepsilon)$-approximate nearest neighbor. Revisiting Lemma 3.5.1, we suppose that $x^{m-\log_5(40/\varepsilon)}$ has been deleted, and suppose that the refinement search considered in its stead a descendant $\tilde{z}^0$. By Property 2 $d(x^{m-\log_5(40/\varepsilon)}, \tilde{z}^0) < \frac{4}{5}5^{m-\log_5(40/\varepsilon)} - 1 = \frac{1}{50}5^m\varepsilon - 1$. It follows that $d(q, \tilde{z}^0) \leq d(q, x^{m-\log_5(40/\varepsilon)}) + d(x^{m-\log_5(40/\varepsilon)}, \tilde{z}^0) < (\frac{1}{50}5^m\varepsilon - 1 + d(q,z)) + (\frac{1}{50}5^m\varepsilon - 1) = \frac{1}{25}5^m\varepsilon - 1 + d(q,z)$ (where z is the nearest neighbor of q in S). $\tilde{z}^0$ is therefore a $(1+\delta)$-approximate nearest neighbor of q, where $1+\delta = \frac{d(q,\tilde{z}^0)}{d(q,z^0)} = \frac{\frac{1}{25}5^m\varepsilon - 1 + d(q,z^0)}{d(q,z^0)} = 1 + \frac{\frac{1}{25}5^m\varepsilon - 1}{d(q,z^0)} < 1 + \frac{\frac{1}{25}5^m\varepsilon - 1}{\frac{1}{5}\cdot 5^{m-1}+1} < 1 + \frac{\frac{1}{25}5^m\varepsilon}{\frac{1}{5}\cdot 5^{m-1}} = 1 + \varepsilon$.

3.7 Centroid Path Updates

In this section, we show how to dynamically maintain the centroid path decomposition under insertions to the hierarchy. Recall that (in Section 3.4.1) we defined a centroid path to be the path starting at the root, which at each node branches to the node's largest child. We further stipulated that each centroid path C_i contains nodes of size $2^j < s(v) \leq 2^{j+1}$ for appropriate i. In a centroid path decomposition, we recursively decompose each off-path subtree of the centroid path.

To support the cover search, we required that each centroid path C_i (or rather the nodes of C_i) be stored in a weighted search structure for C_i, where the weight of a node is the number of nodes in its off-path subtrees, plus one for the node itself. We use the biased skip lists of Bagchi *et al.* [4], but will need to modify these slightly to achieve the insertion time of $O(\log n)$ per new node. Left to right order in the skip lists will correspond to bottom to top order in the centroid path.

We review biased skip lists in the next section (Section 3.7.1), and briefly mention what modifications to the skip lists will be necessary. In Section 3.7.2 we discuss what changes occur to the tree T' and its centroid path decomposition when a point is inserted into the hierarchy, and in Section 3.7.3 we show how to modify the biased skip lists for our purposes.

3.7.1 Review of biased skip lists

A skip list is a data structure which stores an ordered, weighted set X in its ordered nodes. Each element $x_i \in X$ of weight w_i is stored in a skip list item (or node) e_i of *height* at least $\lceil \log w_i \rceil$ ($h(e_i) \geq \lceil \log w \rceil$). If $h(e_i) > \lceil \log w_i \rceil$, then we say that e_i is *oversized*. The height of the skip list, H, is equal to the height of its highest item, and the *depth* of item e_i is the height of the list minus the height of e_i ($H - h(e_i)$). At the left and right ends of the list are sentinel nodes of height H, which do not correspond to points of X.

The items of a skip list are kept in order in a doubly linked list. In addition, there exists a doubly linked list for each value $0 \leq h \leq H$, which stores the ordered items of height *at least* h; this is called the h-list.

The skip list is parametrized by two integer constants a, b with $1 < a \leq \lfloor b/2 \rfloor$. The (a, b)-skip list obeys the following invariants.

Invariant 1. *For all h, $0 \leq h \leq H$, there are at most b consecutive items of height exactly h in the h-list.*

Invariant 2. *For each oversize item e, and for all $\lceil \log w(e) \rceil < h \leq h(e)$, the skip list contains at least a items of height exactly $h - 1$ between e and its h-list neighbors to the right and left.*

Note that Invariant 2 applies only to an oversized item e. The sequence of height $h-1$ items that separates oversized item e from its h-list neighbors will be called the oversize $(h-1)$-sequence of e. If this sequence separates e

from the right or left sentinel, it is called the right or left end $(h-1)$-sequence. If an end sequence is empty, it is a *null sequence*.

Note that these invariants imply that the height of the skip list is $O(\log W)$, (where W is the sum of weights of the items in the list), and that the depth of an item with weight w is $O(\log \frac{W}{w})$.

Operations supported by the biased skip list

Biased skip lists support the operations search, insert, delete, and reweight. Here we briefly sketch these operations, and refer the reader to [4] for a more complete description.

A search is initialized at the left sentinel, and inspects the H-list (which has the left sentinel as its first element). The search locates the item e_i in the H-list that possesses the sought key, or the pair of items e_i, e_{i+1} whose keys straddle the sought key. The search then considers item e_i, and inspects the $(H-1)$-list from e_i and progresses to the right to again find an item that possesses the sought key, or the pair of items whose keys straddle the sought key. The search then traverses down to the next level as before. The search terminates with success if it identifies the item with the sought key, or the final two items which straddle it. Note that only b work is done at every level, and since an item of weight w has depth $O(\log \frac{W}{w})$, the time required to find this item is $O(\log \frac{W}{w})$.

The operation insert inserts an item e of weight w in its ordered position in the structure. Finding this position necessitates a binary search, and can

be done in $O(\log W)$ time. e is given height $h = \lceil \log w \rceil$. All relevant h-lists are updated to include e, in $O(\log w)$ time. The insertion may cause the violation of Invariant 1, if the h-list now contains b consecutive items of height h. This violation is repaired by promoting one of these items to give it greater height, thereby *splitting* the sequence. Similarly, the insertion of e may cause the violation of Invariant 2, if e interrupted a sequence of a consecutive items of height $h' < h$ adjacent to an oversized item e'. In this case e' is demoted until it is not oversized, or until it has height h'. Note that the demotions may cause up to $O(\log W))$ violations at height h' and up, but these can be repaired with $O(\log W)$ work. Similarly, the split repairs can be repaired in a bottom-up fashion in $O(\log W)$ time. It follows that the total cost of the insertion is $O(\log W)$.

The operation delete removes item e of weight w from the skip list. All h-lists containing e are updated, which can be done in $O(\log W)$ time. e may have separated two valid sequences of height $h-1$; the removal of e causes these two sequences to *join*, possibly creating a sequence of length greater than b, and violating Invariant 1. Similarly, the removal of e may cause the violation of Invariant 2 if e was part of a sequence of exactly a consecutive items of height $h-1$ adjacent to an oversized item. As only levels $h-1$ and higher are affected, these violations and subsequent violation at higher levels may all be repaired with $O(\log \frac{W}{w})$ work. The total repair cost is then $O(\log W)$.

The operation reweight changes the weight of an item e from w_1 to w_2.

The reweighting involves updating the h-lists between the initial and final height of e. Similarly, these levels and all higher levels may need to be repaired. Hence, the total cost of the reweighting is $O(\log \frac{W+w_2}{w_m})$, where W is the total weight of the skip list before the reweighting, and $w_m = \min\{w_1, w_2\}$.

In Section 3.7.3, we will modify the biased skip lists so that the cost of inserting an item of weight w at the left end of the list, or deleting an item of weight w from the right end of the list, is only $O(\log w)$, as opposed to $O(\log W)$.

3.7.2 Dynamic changes to T'

The tree T' will change as insertions into the hierarchy occur. A single insertion into the point set may result in the addition of $2^{O(\lambda)}$ nodes in T', which may be leaf nodes or internal nodes. We will consider each one of these $2^{O(\lambda)}$ nodes separately, and show that each node insertion can be supported in $O(\log n)$ time. Hence, a point insertion can be supported in $2^{O(\lambda)} \log n$ time.

The insertion of a node v into a tree entails the insertion of v into the appropriate centroid path. The biased skip list for this path is then updated to include this node in $O(\log W)$ time. The insertion of v causes additional updates up the tree, as $O(\log n)$ ancestors of v gain an extra descendant in their off-path subtrees, and have their weight increase by one. These ancestors must be reweighted. Let the series of such ancestors be $v_1, v_2, \ldots, v_k$ of weight $w_1, w_2, \ldots, w_k$, respectively. Note that the total weight of C_i, the

centroid path of v_i, is less than the weight of v_{i+1} (since v_{i+1} contains C_i in its off-path subtree). We have that the cost of the reweightings is $O(\log(w_1 + 1) + \sum_{i=1}^{k-1} \log \frac{w_{i+1}+1}{w_i+1}) = O(\log w_1) = O(\log n)$.

An outstanding issue is that a node insertion may cause some higher node to become too heavy for its current centroid path; this node will then need to migrate to the centroid path that is the parent of its current centroid path. In fact, a node insertion may cause $O(\log n)$ ancestors to leave their current centroid path for the next path up. Consider one such ancestor v_i of weight w_i that is removed from its centroid path C_i. There will be at least w_i insertions into the subtree rooted at the top of C_i before another node is removed from C_i. Thus we can afford $\Theta(w_i)$ steps to handle the node transfer, though we will show in the next section that $O(\log w_i)$ work suffices for these *end updates*. Note that an end update taking $O(\log w_i)$ time will be performed over the next $\log w_i$ insertions into the relevant subtree of S.

The final concern is to ensure that the search time is not affected when the node transfer is proceeding. But this presents no real difficulty. We will keep the single node "in between" paths while the updates to the centroid paths are computed. When a centroid path is searched, we will add a single query to inspect the nodes between the path and its parent's path, without affecting the asymptotic running time.

Comment. An attempt to use a topology tree [21] for our construction encounters two obstacles in the execution of the search: (i) The transfer to friends. This can be solved by using the (very intricate) dynamic lca query

structure of Cole and Hariharan [19]. (ii) Determining which nodes have descendants storing non-deleted points (see Section 3.6.4); it is not clear if this difficulty can be solved.

3.7.3 Modified biased skip lists

We augment the biased skip lists of Bagchi et al. [4] to support fast end updates. We will show that an item of size w can be inserted into the left end or deleted from the right end of the biased skip list in $O(\log w)$ time. The other run times supported by the biased skip list (search, insert, delete, reweight) remain unchanged. To enable us to achieve this result we will weaken Invariants 1 and 2, but *only* in regards to the portion of an h-list that starts at the left sentinel node or ends at the right sentinel node. The following invariants override Invariants 1 and 2.

Invariant 3. *For all h, $0 \leq h \leq H$, there are at most $b+1$ consecutive items of height exactly h at the left or right ends of the h-list.*

Invariant 4. *For each oversize item e, and for all $\lceil \log w(e) \rceil < h \leq h(e)$, the h-list contains at least $a-1$ items of height exactly $h-1$ between e and the sentinel nodes to the right and left.*

An end update of an item of size w must preserve these invariants, while taking only $O(\log w)$ time. Before detailing the end updates, we will first outline what splits and joins will be necessary to support the end updates. For this purpose, we use splits that take an h-sequence of $b+1$ or $b+2$ items

and split the sequence (by promoting an item in the sequence) to create sequences of sizes $\lfloor b/2 \rfloor$ and $\lceil b/2 \rceil$, or of sizes $\lfloor (b+1)/2 \rfloor$ and $\lceil (b+1)/2 \rceil$, respectively. We will also utilize joins that take h-sequences that contain $a-1$ or $a-2$ items and $a+c$ items ($c \geq 0$, $a+c \leq b$), and join these sequences (by demoting the separating item) to produce an h-sequence of size $2a+c$ or $2a+c-1$ items. (The new sequence includes the demoted item). We may also join h-sequences and immediately split the joined sequence if it is large: A join immediately followed by a split takes the same pair as before and produces h-sequences of $\lceil (2a+c-1)/2 \rceil$ and $\lfloor (2a+c-1)/2 \rfloor$ items, or of $\lceil (2a+c-2)/2 \rceil$ and $\lfloor (2a+c-2)/2 \rfloor$ items, respectively. Later, we will need that the sizes of the new h-sequences lie in the range $[a+1, b-1]$. As the split that follows a join splits a list of size at least b and at most $a+b-1$, it must create two lists of size in the range $[a+1, b-1]$; it suffices that $\lfloor \frac{b-1}{2} \rfloor \geq a+1$ and $\lceil \frac{a+b-2}{2} \rceil \leq b-1$. Likewise, the split of a list of $b+1$ or $b+2$ items creates the additional constraint that $\lceil \frac{b+1}{2} \rceil \leq b-1$. Since $a \geq 2$ (as a join can consider a sequence of size $a-2$), we can choose $a=2$, $b=7$ to fulfill these conditions.

Now that we have described what types of splits and joins will be used, we can proceed to describe the procedure for a left end update, which modifies left end sequences, and the procedure for right end updates, which modify right end sequences. But first we need to specify some more constraints observed by the end h-sequences. A left or right end h-sequence σ can occupy one of five states: -2,-1,0,1,2. It is in state -2 if the next item e in the h-list

is oversize and σ holds $a-1$ items, in state -1 if e is oversize and σ holds a items, in state 1 if σ holds b items, in state 2 if σ holds $b+1$ items, and in state 0 otherwise. (Note that the goal of Invariants 3 and 4 was to allow for these five states.)

We keep two stacks, one for each sentinel node. The left sentinel stack stores pointers to all left end h-sequences with the stack order from bottom to top corresponding to decreasing h-index order. The right sentinel stack is organized analogously.

When an item is added to the end h-sequence in state $c < 2$, the state of the sequence changes to $c+1 \leq 2$ (or possibly remains at 0 if c was 0). Then the h-sequence does not violate Invariant 3, and does not need to be repaired. When an item is added to the end h-sequence in state 2, the sequence violates Invariant 3 and must be repaired. The sequence splits, with the result that the sequence is now in state 0, and that one item has been promoted to the end $(h+1)$-sequence. Crucially, we will demonstrate below that the $(h+1)$-sequence was not previously in state 2 (that is, there are no consecutive state 2 sequences in the stack), which means that the promoted item does not cause the $(h+1)$-sequence to now violate Invariant 3, and no further repairs are necessary.

When an item is removed from an end h-sequence in state $c > -2$, the state of the sequence becomes $c-1 \geq -2$ (or possibly remains at 0 if c was 0). Then the h sequence does not violate Invariant 4, and does not need to be repaired. When an item is removed from an end h-sequence in

state -2, then the sequence violates Invariant 4 and must be repaired. Note that immediately on the inside of the sequence there is an oversize item e of height $h+1$. e is demoted to height h. The two h-sequences on either side of e are then joined, or if need be joined and immediately split. In the former case this results in a state 0 end h-sequence, and the demotion of e reduces the index of the end $(h+1)$-sequence by 1 (or if it was at state 0 may leave its state unchanged). Crucially, we will demonstrate below that the $(h+1)$-sequence was not previously in state -2 (that is, there are no consecutive state -2 sequences in the stack), which means that the demotion of e does not cause the $(h+1)$-sequence to now violate Invariant 4, and no further repairs are necessary. In the latter case (a join followed by a split), this results in a state 0 end h-sequence, while the $(h+1)$-sequence remains unchanged (since it lost the item e and gained another item due to the split).

It is left to ensure that there are no consecutive state 2 or state -2 sequences in the stack. This will be achieved by the following state distribution invariant.

Invariant 5. *(i) State 2 sequences on the stack are separated by at least one sequence which has state $c \leq 0$ or which is null. Similarly, state -2 sequences on the stack are separated by at least one sequence which has state $c \geq 0$ or which is null.*

(ii) The state 2 sequence with the lowest index in the left stack is preceded by a sequence which has state $c \leq 0$, or which is null. Similarly, the

state -2 sequence with the lowest index in the right stack is preceded by a sequence which has state $c \geq 0$, or which is null.

Prior to adding an item e of height $\lceil \log w \rceil$ to the left end of the skip list, we bring all end h-sequences for $h \leq \lceil \log w \rceil$ to state -1,0, or 1 by performing splits and joins on the sequences in increasing h order. In addition, σ, the next end h-sequence in state ± 2 (in increasing h order), if any, is also brought to state 0. Following this, item e is added to the end $\lceil \log w \rceil$-sequence, and if this sequence is now in state 2 it is split. Note that all previous end h-sequences for $h < \lceil \log w \rceil$ cease to be end sequences, but as they are in state -1,0, or 1 they obey Invariants 1 or 2 as appropriate.

Prior to the deletion of an item e of height $\lceil \log w \rceil$ from the right end of the skip list, there are only null end h-sequences for $h < \lceil \log w \rceil$. The sequences that become the end h-sequences for $h < \lceil \log w \rceil$ following the deletion are therefore all in state -1,0, or 1, as they obey Invarants 1 and 2. Prior to the deletion, the end $\lceil \log w \rceil$-sequence is brought to state -1,0, or 1 (if not already there), and the next higher end h-sequence in state ± 2 (if any) is brought to state 2. Now item e is removed.

Lemma 3.7.1. *The procedure maintains Invariants 1–5.*

Proof. We start by considering an insertion of item e of weight w. We begin by claiming that prior to adding e to the $\lceil \log w \rceil$-sequence, the lowest state ± 2 h-sequence (if any) is preceded by an h'-sequence, $h' < h$, in state 0 with $h' > \lceil \log w \rceil$. It then follows that when e is added, Invariant 5(i) holds.

If the $\lceil \log w \rceil$-sequence is then split, Invariant 5(i) still holds, and the new $\lceil \log w \rceil$-sequence, in state 0, restores Invariant 5(ii). To see the claim, note that the state ± 2 h-sequence ($h > \lceil \log w \rceil$) that was brought to state 0 prior to e's addition provides one state 0 sequence between the $\lceil \log w \rceil$-sequence and the next state ± 2 sequence.

That Invariants 1–4 continue to hold is clear from the description of the splits and joins.

Exactly the same reasoning shows that the deletion procedure maintains the invariants.

□

Comment. We conjecture that essentially this construction can be applied to the biased search trees of Bent *et al.* [6].

The next issue to mention is what happens when a reweighting or non-end update performs a split or join on an end sequence. This is performed as usual, with the corresponding sequence being updated in the stack. Invariant 5 continues to hold, as in the proof of Lemma 3.7.1.

Theorem 3.7.2. *The modified biased skip list structure stores a set of weighted ordered items supporting searches in* $O(\log \frac{W}{w})$ *time, reweightings in* $O(\log \frac{W}{w_m})$ *time, and end updates in* $O(\log w)$ *time.*

3.8 Linear space and efficient storage of friends lists

We have previously shown that the number of hierarchical points appearing explicitly is $O(n)$: Lemma 3.3.1 demonstrates that only $O(n)$ points have friends, so $O(n)$ points appear explicitly due to their having friends. Further, the insertion scheme stipulates that an insertion or promotion adds $O(1)$ explicit points that lack friends. This accounts for $O(n)$ points.

However, we have stipulated that we store the friends list of each point, and this requires $2^{\Theta(\lambda)}n$ space, which is superlinear. To this end, we must relax the condition that all friends lists be stored. Instead, only some points will have their friends list stored.

Suppose that we knew λ, the doubling dimension of the space. We know then the maximum number of friends that any ball may possess; this quantity is $2^{O(\lambda)}$, or $2^{\lambda'}$ for brevity. If we keep friends lists for only $n/2^{\lambda'}$ points then the resulting structure uses linear space. Since there are $O(n/2^{\lambda})$ points at height λ or greater in the modified biased skip lists, it suffices to store friends lists for only these points.

As we do not know λ', instead we limit the length of a point's friends list according to the height in the skip list of the node that stores the point; a point stored by a node at height $2i$ or $2i+1$ will be allowed to store a list of length at most 2^i. Any point that has more friends than can fit in its list is marked as *incomplete*. Again, this uses linear space.

The search proceeds as before, until we reach a node whose point is marked incomplete. In this scenario, we backtrack in the biased skip list until we reach a level at which all the nodes being examined store points with complete friends lists; that is, at most up to height $2\lambda'$. We then proceed with the containment search by descending λ' levels in a manner similar to the search procedure of [28] (described in Section 2.2.2): At each level the search maintains $2^{O(\lambda)}$ nodes that store points that cover q, inspects all children of these nodes, and takes the children that store points that cover q. The search time increases by an additive term of $\lambda' 2^{O(\lambda)} = 2^{O(\lambda)}$. We conclude that:

Theorem 3.8.1. *The data structure may be implemented in $O(n)$ space.*

Chapter 4

Application: A spanner

Now that we have fully described the hierarchy of Kraughgamer and Lee [28], as well as the new hierarchy, we move to creating a spanner for the point set S. In Section 4.1, we will build a (crude) spanner using the old hierarchy. In the next section (Section 4.2) we will build on these techniques to create a better spanner.

4.1 A first-attempt spanner

The hierarchy described in Section 2 can be used as a backbone for a geometric spanner. A few definitions are necessary. First recall that for a spanner H, $d_H(x, y)$ is the spanner distance between x and y. A point $x \in Y_{2^i}$ is a parent of $y \in Y_{2^{i-1}}$ if x covers y. If more than one point covers y then the closest one of these points is chosen to be the parent of y. A point x is an

ancestor of y if there exists a series of points $\langle x, \ldots, y \rangle$ such that each point in the series is a parent of the subsequent one.

We use the hierarchical partition to decide which edges are included in the spanner. There are two types of edges. The first type consists of *parent-child edges* that connect each point in Y_{2^i} to its parent in $Y_{2^{i+1}}$. A point in $Y_{2^{i+1}}$ may have $2^{O(\lambda)}$ children, so this adds $2^{O(\lambda)}$ child-parent edges for each occurrence of a point in the hierarchy. The second type consists of *lateral edges* which connect points in the same level when the distance between them is below some threshold. Specifically, in level Y_{2^i} we add an edge between any two points that are within distance $c \cdot 2^i$ (for some constant c that will depend on the desired precision ε). A point in level Y_{2^i} may have $c^{O(\lambda)}$ points within distance $c \cdot 2^i$, so this adds $c^{O(\lambda)}$ lateral edges for each occurrence of a point in the hierarchy.

Let H be a spanner that contains the aforementioned parent-child and lateral edges, and let $c \geq 16(\frac{1}{\varepsilon} + \frac{3}{4})$. We can show that H has low stretch. Before proving this, we note a simple property of the hierarchy:

Property 3. *Let $x' \in Y_{2^i}$ be an ancestor of $x \in Y_{2^j}$ $(i > j)$. Then $d_H(x, x') \leq \sum_{k=j+1}^{i} 2^k = 2 \cdot 2^i - 2^{j+1} < 2 \cdot 2^i$.*

The main lemma of this section follows.

Lemma 4.1.1. *H is a $1 + \epsilon$ spanner for S.*

Proof. We must show that for any two points $x, y \in Y_1$, $\frac{d_H(x,y)}{d(x,y)} < 1 + \frac{1}{\frac{c}{16} - \frac{3}{4}}$.

If $d(x,y) \leq c$ then x and y are connected by a lateral edge, so $d_H(x,y) = d(x,y)$ and we are done.

Otherwise, let $x', y' \in Y_{2^i}$ be the lowest ancestors of x and y (respectively) which are connected by a lateral edge. Since x' and y' are connected by a lateral edge $d_H(x',y') = d(x',y')$. Also note that by Property 3, $d_H(x,x')$ and $d_H(y,y')$ are both less than $2 \cdot 2^i$, from which it follows that $d(x,x')$ and $d(y,y')$ are also less than $2 \cdot 2^i$.

Now, the spanner distance from x to y is $d_H(x,y) \leq d_H(x,x') + d_H(x',y') + d(y',y) < 2 \cdot 2^i + d(x',y') + 2 \cdot 2^i = d(x',y') + 4 \cdot 2^i$. The true distance from x to y is $d(x,y) \geq d(x',y') - d(x',x) - d(y',y) > d(x',y') - 2 \cdot 2^i - 2 \cdot 2^i = d(x',y') - 4 \cdot 2^i$. It follows that the stretch of the spanner is less than $\frac{d_H(x,y)}{d(x,y)} = \frac{d(x',y')+4\cdot 2^i}{d(x',y')-4\cdot 2^i} = 1 + \frac{8\cdot 2^i}{d(x',y')-4\cdot 2^i}$. This term reaches its maximum when $d(x',y')$ reaches its minimum.

It remains only to place a lower bound on $d(x',y')$. By assumption, the children of x' and y' on the paths to x and y (in level $Y_{2^{i-1}}$) are not connected by a lateral edge, so the distance between them is greater than $c \cdot 2^{i-1}$. The distance from x' to its child (and the distance from y' to its child) is less than 2^i. It follows that $d(x',y') > c \cdot 2^{i-1} - 2^i - 2^i = c \cdot 2^{i-1} - 2 \cdot 2^i$.

The stretch of the spanner is less than $1 + \frac{8\cdot 2^i}{d(x',y')-4\cdot 2^i} < 1 + \frac{8\cdot 2^i}{c\cdot 2^{i-1}-6\cdot 2^i} = 1 + \frac{1}{\frac{c}{16}-\frac{3}{4}}$. Choosing $c \geq 16(\frac{1}{\varepsilon} + \frac{3}{4})$ yields a $(1+\varepsilon)$-spanner. □

Since a point may appear in $O(\log \alpha)$ levels of the hierarchy, it may have $c^{O(\lambda)} \log \alpha = O(\frac{\log \alpha}{\varepsilon^{O(\lambda)}})$ lateral edges incident upon it in the spanner, and so the degree of the spanner is $O(\frac{\log \alpha}{\varepsilon^{O(\lambda)}})$.

To understand why the spanner construction guarantees low stretch, note that the path from two points $x, y \in Y_1$ consists of a set of parent-child edges, followed by a single lateral edge, followed by a second set of parent-child edges. Choosing a large value for c causes the length of the lateral edge to dwarf the lengths of the other edges, and this results in $d_H(x, y)$ being close to $d(x, y)$.

We have already reviewed in Section 2.4 how the hierarchical partition of [28] can be maintained dynamically in $O(\frac{\log \alpha}{\varepsilon^{O(\lambda)}})$ update time, and it is an easy matter to maintain the aforementioned spanner in the same time as well.

4.2 The new spanner

In this section, we build upon the new hierarchy of Section 3 to create a $(1+\varepsilon)$-spanner with degree $(1/\varepsilon)^{O(\lambda)}$. This spanner can be maintained dynamically in $O(\frac{\log n}{\varepsilon^{O(\lambda)}})$ time; we defer a discussion of dynamic updates to Section 4.2.7. We assume that we have access to the hierarchy and its associated spanning tree T. (In describing the spanner, it will be easier to refer to the uncompressed tree T as opposed to the compressed tree T'.)

4.2.1 Motivation: An incremental spanner.

Suppose for the moment that we wished to maintain a spanner under insertions alone, so that the hierarchy contained no deleted points. Then it would be possible to maintain a dynamic spanner in $O(\frac{\log n}{\varepsilon^{O(\lambda)}})$ time using the new

hierarchy as a backbone. The spanner is created by assigning parent-child and lateral edges to all points.

This construction guarantees low stretch: As before, the path from two points x^0 and y^0 at the bottom of the hierarchy consists of a set of parent-child edges from x^0 up to one of its ancestors, followed by a single lateral edge to an ancestor of y^0, followed by a second set of parent-child edges down to y^0. Choosing a large value for the bound on the size of lateral edges causes the length of the lateral edge to dwarf the lengths of the other edges, and this results in the spanner having low stretch. (We omit the exact analysis, which is similar to what was shown in the previous section.)

The difficulty with this approach is that the hierarchy contains deleted points which cannot appear in the spanner. Further, since a point may appear in many levels of the hierarchy, and possess lateral edges for each level, the degree of the spanner may be very large. Below, we will use the spanning tree T to create a new hierarchy that addresses both of these problems: The new hierarchy contains no deleted points, and each point appears in the hierarchy at most three times (once in the bottom level and up to twice more in higher levels). We will use this new hierarchy to create a spanner that mimics the spanner described above.

Let the hierarchy, tree and spanner described above be called the *full* hierarchy, tree and spanner. Below, we present the new hierarchy in two steps. The first step (in Section 4.2.2) prunes nodes of T to create a smaller tree T^1. The points stored in the nodes of T^l are a subset of the points of the

full hierarchy. We call T^1 the *intermediate* tree, and the points stored in T^1 constitute the intermediate hierarchy. We then give a scheme that replaces points stored in the internal nodes of T^1 by non-deleted points. The *final* tree T^2 (presented in Section 4.2.3) is the tree T^1 after the point replacement. T^1 stores the final hierarchy, which contains no deleted points, and in which each point appears at most three times.

4.2.2 Step 1. Pruning the spanning tree

Recall that each node of tree T stores a point of the full hierarchy. The first step in creating a new hierarchy involves pruning T in a straightforward manner, thereby creating a new spanning tree T^1 which stores fewer points.

Let *real* nodes (or leaves) in T be nodes that store non-deleted points, and Steiner nodes (or leaves) be those nodes that store deleted points (which are known as *Steiner points*). We create T^1 from T in two steps: First, we remove from T all Steiner leaves, as well as all nodes that have no real leaf descendant. Then we compress all single-child paths. (A single-child path is a maximal chain of parent-child nodes where each node, including the final one, has only one child. In compressing the path we retain only the first node, and link it to the single child of the final node.) The resulting tree is T^1. By construction, parent-child relationships in T^1 may have been ancestor-descendant relationships in T. Also, all remaining internal nodes have at most $2^{O(\lambda)}$ children, and have real leaf node descendants. Each node either has at least two children, or has a child with at least two children.

(For a node $v \in T$ that *survives* in T^1, we may refer both to $v \in T$ and to $v \in T^1$.)

The nodes of T store points in the full hierarchy, and so the nodes of T^1 (which are a subset of the nodes of T) store a subset of points of the full hierarchy. We will call this subset of points the *intermediate* hierarchy. Because the construction of T^1 compressed single-child paths, the intermediate hierarchy obeys the packing property but not the covering property. However, the presence of parent-child connections between points in different levels of the hierarchy implies that it does obey a somewhat weaker covering property, where every point in level Y_{5^i} is strictly within the radius of some point residing in a higher level (but not necessarily in level $Y_{5^{i+1}}$).

4.2.3 Step 2. Creating a better hierarchy

A slight modification to the intermediate hierarchy will yield the *final* hierarchy with the properties we want: It contains no deleted points, and each point appears in at most three levels.

Recall that T^1 contains no Steiner leaves, but may contain other Steiner nodes. Since Steiner nodes store deleted points, the intermediate hierarchy contained deleted points. We will create from T^1 a tree T^2 that stores the final hierarchy. We first introduce the following *assignment* scheme to associate each internal node with a leaf node: Assume an arbitrary left-right ordering on the children of internal nodes. As an outcome of the single-path compression, each internal node of T^1 either has at least two children or has

a child that has at least two children. This means that there are fewer than two internal nodes for each leaf node, which allows us to assign a leaf node to at most two ancestral internal node. For example, to each internal node v with at least two children, we assign to v the leftmost leaf descendant of v's rightmost child. To each internal v node with one child, we assign to v the same leaf node assigned to its child. Note that this scheme assigns a unique leaf node to each internal node that has at least two children, and a unique leaf node to each internal node that has one child. Hence, each leaf node is assigned to at most two ancestral internal nodes.

T^2, the tree that stores the final hierarchy, is created from T^1 using the assignment scheme. T^2 is initialized as an exact copy of T^1 (along with its assignment scheme). Then, for each internal node on T^2, we replace its stored point by the point stored in its assigned leaf node descendant. T^2 now stores the points of the final hierarchy. The replacement step that creates the final hierarchy can be viewed as removing each point y^l in the intermediate hierarchy and replacing y^l by a descendant z^0. By the close-covering property, $d(y, z) < \frac{4}{5} \cdot 5^l - 1$; hence x and y are relatively 'close', and the final hierarchy can be viewed as a minor perturbation of the intermediate hierarchy. Crucially, the final hierarchy contains no deleted points, and each point appears in at most three levels.

Now that we have derived the final hierarchy and its spanning tree T^2, we can use it to extract a spanner. For presentation purposes, we will first give a spanner for the intermediate hierarchy (which contains Steiner points), since

this spanner is more intuitive. The spanner for the final hierarchy is almost identical to the spanner of the intermediate hierarchy, only with the points of the intermediate hierarchy replaced by their descendants according to the assignment scheme.

4.2.4 Step 3. A spanner for the intermediate hierarchy

We wish to construct a spanner for the intermediate hierarchy; the new spanner should resemble the full spanner, and have the equivalent of parent-child edges and lateral edges. As before, the length of the lateral edges dwarf the lengths of parent-child edges, resulting in a spanner with low stretch.

Type I edges. The new spanner will have edges that mimic the behavior of parent-child edges in the full spanner.

Consider node v that survives in T^1. v has a parent node in T, and in the full spanner the points stored by these nodes have a parent-child edge connecting them. v's parent in T^1 was the parent or ancestor of v in T, and the points they store are an ancestor-descendant pair in T, and a parent-child pair in T^1. We add a spanner edge between these two points; this is a *parent-child edge* for the intermediate spanner.

We will need another type of edge to make up for the fact that the intermediate hierarchy obeys only a weak covering property. (That is, a point may be covered by another point at a much higher level). This new edge is similar to a parent-child edge: Let x^k and z^m $(k > m)$ be two points in the intermediate hierarchy that are an ancestor-descendant pair. Let y^l $(l > m)$

be the lowest point in the hierarchy that covers z^m (or, if many points in that level cover z^m, an arbitrary one of these). If y^l is below x^k (that is, $k > l$), then y^l becomes the step-parent of z^m. We add a spanner edge between y^l and z^m; this is a *step-parent edge*.

(We will see in Section 4.2.6 that step-parent edges are key to attaining low spanner degree. Note also that we did not specify how y can be located; we defer a description of this to Section 4.2.7.)

Type II edges. The new spanner will have edges that mimic the behavior of lateral edges in the full spanner.

Consider node v that survives in T^1. The point in the hierarchy stored in v, say y^l, is present in the intermediate hierarchy, and was also found in the full hierarchy. In the full spanner, y^l possessed a lateral edge to all level Y_{5^l} points within distance $c \cdot 5^l$ of y^l; Let R be the set of these points. For each point $x^l \in R$ that is present in the intermediate hierarchy, we add to the new spanner a lateral edge connecting x to y. Now, for each point $x^l \in R$ that is not present in the intermediate hierarchy, we must find an equivalent for the now missing lateral edge from x^l to y^l: Let z^m be the highest descendant of x^l (in the full hierarchy) that is still present in the intermediate hierarchy. We add a *replacement lateral edge* from y^l to z^m.

However, there is an exception to this rule: If z^m has a step-parent in level Y_{5^l} or lower, then we will not replace the missing lateral edge from x^l to y^l. The intuition behind this exception is that there now exists an alternate path from z^m up to Y_{5^l} or a lower level, so with respect to z^m it is not necessary

to replace the missing lateral edge. (We will see in Section 4.2.6 that this step-parent exception is key to attaining low spanner degree.)

Finally, for all point pairs x and y connected by a lateral edge or by a replacement lateral edge, we add *lateral refinement edges* between x and the children of y, between y and the children of x, and between the children of x and y. The purpose for this will become clear in the proof of Theorem 4.2.1 immediately below.

Proof of low stretch

Let H be the spanner for the intermediate hierarchy described above (where $c \geq \frac{25}{\varepsilon} + \frac{5}{2}$); H is a spanner for S, but uses Steiner points.

Theorem 4.2.1. *H is a $(1+\varepsilon)$-spanner for S.*

Proof. To prove this, we define the notion of an *ancestral path* from a point y^0 in the intermediate hierarchy towards some level Y_{5^m}. The path begins at y^0, and at each step proceeds to the current point's step-parent. If the current point has no step-parent, then the path proceeds to its parent in the intermediate hierarchy. The path terminates when the next candidate point is above level Y_{5^m}. The spanner distance (and true distance) from y^0 to any other point x^p on the path is less than $5^p \sum_{i=0}^{\infty} (\frac{1}{5})^i = \frac{5}{4} 5^p \leq \frac{5}{4} 5^m$.

Let x^0, y^0 be any two points at the bottom level of the hierarchy. We will show that $\frac{d_H(x^0,y^0)}{d(x^0,y^0)} < 1+\varepsilon$. First define m by $(c - \frac{5}{2}) \cdot 5^{m-1} \leq d(x,y) < (c - \frac{5}{2}) \cdot 5^m$. Let the last node in the ancestral path from x^0 (y^0) towards level

Y_{5^m} be x' (y'). We will show below that x' and y' are connected by a lateral edge in the spanner (that is, x' and y' are connected either by a lateral edge, a lateral replacement edge, or a lateral refinement edge); this implies that there is a path from x^0 to y^0 which consists of the edges between points on the ancestral path from x^0 to x', followed by a single lateral edge from x' to y', followed by the edges between points on the ancestral path from y' to y^0. As before, the length of the lateral edge dwarfs that of the other edges, resulting in low stretch.

More rigorously: We know that $d(x', y') \leq d(x^0, y^0) + d(x^0, x') + d(y^0, y') < d(x, y) + \frac{5}{4}5^m + \frac{5}{4}5^m = d(x, y) + \frac{5}{2}5^m$. It is also true that $d_H(x^0, y^0) \leq d_H(x', y') + d_H(x', x^0) + d_H(y', y^0) < d_H(x', y') + \frac{5}{4}5^m + \frac{5}{4}5^m = d_H(x', y') + \frac{5}{2}5^m$. We will show below that there is a lateral edge between points x' and y', so $d_H(x', y') = d(x', y')$. It follows immediately that $d_H(x^0, y^0) \leq d_H(x', y') + \frac{5}{2}5^m = d(x', y') + \frac{5}{2}5^m < d(x^0, y^0) + \frac{5}{2}5^m + \frac{5}{2}5^m = d(x^0, y^0) + 5 \cdot 5^m$. Therefore, the stretch of the spanner is $\frac{d_H(x^0, y^0)}{d(x^0, y^0)} < \frac{d(x^0, y^0) + 5 \cdot 5^m}{d(x^0, y^0)} = 1 + \frac{5 \cdot 5^m}{d(x^0, y^0)}$. This term is maximized when $d(x^0, y^0)$ assumes its minimum possible value, which was defined above to be $(c - \frac{5}{2}) \cdot 5^{m-1}$. It follows that the spanner has stretch less than $1 + \frac{5 \cdot 5^m}{(c - \frac{5}{2}) \cdot 5^{m-1}} = 1 + \frac{25}{c - \frac{5}{2}}$. Taking $c \geq \frac{25}{\varepsilon} + \frac{5}{2}$ yields a $(1 + \varepsilon)$-spanner.

It remains only to demonstrate that x' and y' are indeed connected by a lateral edge in the spanner. Now, if x' and y' are both found in level Y_{5^m}, we have that $d(x', y') \leq d(x^0, y^0) + d(x^0, y') + d(y^0, y') < (c - \frac{5}{2})5^m + \frac{5}{4}5^m + \frac{5}{4}5^m = c \cdot 5^m$, and so x' and y' are connected by a lateral edge. Otherwise, one or both of x' and y' are found in a level below Y_{5^m}. In this case, let $x'' \in Y_{5^l}$ be the

step-parent of x' in the intermediate spanner (or x''s parent if x' has no step-parent), and let $y'' \in Y_{5^k}$ be the step-parent of y' in the intermediate spanner (or y''s parent if y' has no step-parent). Assume without loss of generality that $k \geq l$, and note that since x'' and y'' are not in the ancestral path, $l > m$. If $k = l$, then we have that $d(x'', y'') \leq d(x^0, y^0) + d(x^0, x'') + d(y^0, y'') \leq (c - \frac{5}{2})5^m + \frac{5}{4}5^l + \frac{5}{4}5^l < (c - \frac{5}{2})5^l + \frac{5}{2}5^l = c \cdot 5^l$; in this case, x'' and y'' are connected by a lateral edge, and so x' and y' are connected by a lateral refinement edge. If $k > l$, then let $\widetilde{y}''$ be the ancestor of y' in level Y_{5^l} of the full hierarchy. ($\widetilde{y}''$ did not survive in the intermediate hierarchy.) By the same argument as above, $d(x'', \widetilde{y}'') < c \cdot 5^l$. It follows that in the full spanner, x'' and $\widetilde{y}''$ are connected by a lateral edge, and so in the intermediate spanner, x'' and y' (the child of $\widetilde{y}''$) are connected by a replacement lateral edge. Hence, x' and y' are connected by a lateral refinement edge. □

4.2.5 Step 4. A spanner for the final hierarchy

The spanner for the final hierarchy is similar to the one for the intermediate hierarchy, only with points of the intermediate hierarchy replaced by their assigned points. Recall that the replacement scheme replaces a point in level Y_{5^m} by some descendant, and distance from the original point to any descendant is less that $\frac{4}{5}5^m$. In the intermediate spanner, points x^m and y^m were connected by a lateral edge if $d(x^m, y^m) \leq c \cdot 5^m$. The replacement scheme implies that in the final spanner, we can only make a weaker guarantee: Points x^m and y^m are connected by a lateral edge if

$d(x^m, y^m) \leq c \cdot 5^m - \frac{4}{5}5^m - \frac{4}{5}5^m = (c - \frac{8}{5}) \cdot 5^m$. In the intermediate spanner, the distance from x^m to its step-child z^p was less than 5^m. In the final spanner, the distance from x^m to z^p is less than $(1 + \frac{4}{5})5^m + \frac{4}{5}5^p < 2 \cdot 5^m$. (This upper bounds the distance from a point to its child as well.)

Let H be the spanner for the final hierarchy (where $c \geq \frac{50}{\varepsilon} + \frac{33}{5}$); H is a spanner for S, and does not use Steiner points.

Theorem 4.2.2. *H is a $(1+\varepsilon)$-spanner for S.*

Proof. The proof is similar to the proof of Lemma 4.2.1. As before, an ancestral path towards level Y_{5^m} in the final hierarchy begins at y^0, proceeds to the current point's step-parent (or parent), and terminates when the next candidate point is above level Y_{5^m}. The final spanner distance (and true distance) from y^0 to any other point x^p on the path is less than $2 \cdot 5^p \sum_{i=0}^{\infty} (\frac{1}{5})^i = 2 \cdot \frac{5}{4}5^p = \frac{5}{2}5^p < \frac{5}{2}5^m$.

Let x^0, y^0 be any two points at the bottom level of the hierarchy. We will show that $\frac{d_H(x^0,y^0)}{d(x^0,y^0)} < 1 + \varepsilon$. First define m by $(c - \frac{33}{5}) \cdot 5^{m-1} \leq d(x, y) < (c - \frac{33}{5}) \cdot 5^m$. Let the last point in the ancestral path from x^0 (y^0) towards level Y_{5^m} be x' (y'). We will show below that x' and y' are connected by a lateral edge in the spanner (that is, x' and y' are connected either by a lateral edge, a lateral replacement edge, or a lateral refinement edge); this implies that there is a path from x^0 to y^0 which consists of the edges between points on the ancestral path from x^0 to x', followed by a single lateral edge from x' to y', followed by the edges between points on the ancestral path from y' to

y^0. As before, the length of the lateral edge dwarfs that of the other edges, resulting in low stretch.

More rigorously: We know that $d(x', y') \leq d(x^0, y^0) + d(x^0, x') + d(y^0, y') \leq d(x, y) + \frac{5}{2}5^m + \frac{5}{2}5^m = d(x, y) + 5 \cdot 5^m$. It is also true that $d_H(x^0, y^0) \leq d_H(x', y') + d_H(x', x^0) + d_H(y', y^0) < d_H(x', y') + \frac{5}{2}5^m + \frac{5}{2}5^m = d_H(x', y') + 5 \cdot 5^m$. We will show below that there is a lateral edge between points x' and y', so $d_H(x', y') = d(x', y')$. It follows immediately that $d_H(x^0, y^0) \leq d_H(x', y') + 5 \cdot 5^m = d(x', y') + 5 \cdot 5^m \leq d(x^0, y^0) + 5 \cdot 5^m + 5 \cdot 5^m = d(x^0, y^0) + 10 \cdot 5^m$. Therefore, the stretch of the spanner is $\frac{d_H(x^0, y^0)}{d(x^0, y^0)} < \frac{d(x^0, y^0) + 10 \cdot 5^m}{d(x^0, y^0)} = 1 + \frac{10 \cdot 5^m}{d(x^0, y^0)}$. This term is maximized when $d(x^0, y^0)$ assumes its minimum possible value, which was defined above to be $(c - \frac{33}{5}) \cdot 5^{m-1}$. It follows that the spanner has stretch less than $1 + \frac{10 \cdot 5^m}{(c - \frac{33}{5}) \cdot 5^{m-1}} = 1 + \frac{50}{c - \frac{33}{5}}$. Taking $c \geq \frac{50}{\varepsilon} + \frac{33}{5}$ yields a $(1 + \varepsilon)$-spanner.

It remains only to demonstrate that x' and y' are indeed connected by a lateral edge in the spanner. Now, if x' and y' are both found in level Y^{5^m}, we have that $d(x', y') \leq d(x^0, y^0) + d(x^0, y') + d(y^0, y') < (c - \frac{33}{5})5^m + \frac{5}{2}5^m + \frac{5}{2}5^m = (c - \frac{8}{5}) \cdot 5^m$, and so x' and y' are connected by a lateral edge. Otherwise, one or both of x' and y' are found in a level below Y_{5^m}. In this case, let $x'' \in Y_{5^l}$ be the step-parent of x' in the intermediate spanner (or x''s parent if x' has not parent), and let $y'' \in Y_{5^k}$ be the step-parent of y' in the intermediate spanner (or y''s parent if y' has not parent). Assume without loss of generality that $k \geq l$, and note that since x'' and y'' are not in the ancestral path, $l > m$. If $k = l$, then we have that $d(x'', y'') \leq d(x^0, y^0) + d(x^0, x'') + d(y^0, y'') <$

$(c-\frac{33}{5})5^m + \frac{5}{2}5^l + \frac{5}{2}5^l < (c-\frac{33}{5})5^l + 5\cdot 5^l = (c-\frac{8}{5})5^l$; in this case, x'' and y'' are connected by a lateral edge, and so x' and y' are connected by a lateral refinement edge. If $k > l$, then let $\widetilde{y}''$ be the ancestor of y' in level Y_{5^l} of the full hierarchy. ($\widetilde{y}''$ did not survive in the intermediate hierarchy.) By the same argument as above, $d(x'', \widetilde{y}'') < (c-\frac{8}{5})\cdot 5^l$. It follows that in the full spanner, x'' and $\widetilde{y}''$ are connected by a lateral edge, and so in the intermediate spanner, x'' and y' (the child of $\widetilde{y}''$) are connected by a replacement lateral edge. Hence, x' and y' are connected by a lateral refinement edge. □

4.2.6 The degree of the final spanner

In this section, we prove that the spanner for the final hierarchy has degree $c^{O(\lambda)} = (1/\varepsilon)^{O(\lambda)}$. In proving low degree for the spanner of the final hierarchy, it will be useful to refer back to the spanner of the intermediate hierarchy. Before beginning the proof, we will need an important structural lemma for the intermediate hierarchy. Recall that a point x^j b-covers a point z^m $(j > m)$ if and only if $d(z^m, x^j) \leq b \cdot 5^j$.

Structural lemma

Lemma 4.2.3. *Let w^i and z^m $(i > m)$, be a parent-child pair in the intermediate hierarchy, and $b > 1$ be a parameter.*

(i) If z^m has a step-parent y^j $(i > j > m)$, then there exist only $b^{O(\lambda)}$ points of the intermediate hierarchy in levels Y_{5^j} down to $Y_{5^{m+1}}$ that

b-cover z^m.

(ii) If z^m has no step-parent, then there exist only $b^{O(\lambda)}$ points of the intermediate hierarchy in levels Y_{5^i} down to $Y_{5^{m+1}}$ that b-cover z^m.

Proof. For case (i) let $k = j-1$, and for case (ii) let $k = i-1$. First note that only $b^{O(\lambda)}$ points in each individual level of the intermediate hierarchy may b-cover z^m. Hence, levels $Y_{5^{k+1}}$ and Y_{5^k} contain only $b^{O(\lambda)}$ points that b-cover z^m. It therefore suffices to prove the lemma for points in levels $Y_{5^{k-1}}$ down to $Y_{5^{m+1}}$. To this end, consider the set B of all level Y_{5^k} points in the *full* hierarchy that b-cover z^m; $|B| = 2^{O(\lambda)}$. The proof of the lemma utilizes B, and follows in two steps: We first show that any point in levels $Y_{5^{k-1}}$ down to $Y_{5^{m+1}}$ of the full hierarchy that b-covers z^m must be a descendant of a point in B. This implies that it is sufficient to consider only descendants of B. We then show that each point of B possesses only $b^{O(\lambda)}$ descendants that both b-cover z^m and also survive in the intermediate hierarchy. This implies that only $b^{O(\lambda)}$ points of the intermediate hierarchy b-cover z^m.

We will first show that, as a consequence Property 2 (the close-covering property), any point of the full hierarchy in levels $Y_{5^{k-1}}$ through $Y_{5^{m+1}}$ that b-covers z^m must be a descendant of some point in B (or is itself in B): Let x^l $(k > l > m)$ be a point in the full hierarchy that b-covers z^m, $d(x^l, z^m) < b \cdot 5^l$. By Property 2, the distance from x^l to its full hierarchy ancestor in level Y_{5^k} is less than $\frac{4}{5}5^k - 5^l$. It follows that the distance from z^m to this ancestor is less than $\frac{4}{5}5^k - 5^l + b \cdot 5^l = \frac{4}{5}5^k + (b-1) \cdot 5^l < b \cdot 5^k$, which implies that the

ancestor b-covers z^m.

We will now show that each point in B possesses only $b^{O(\lambda)}$ descendants which both b-cover z^m and also survive in the intermediate hierarchy. Consider a point $x^k \in B$. Suppose that x^k survives in the intermediate hierarchy. Since x^k was not assigned as the step-parent of z^m, and z^m's step-parent or parent is above x^k, it must be that x^k does not cover z^m. Recall that all descendants of x^k are strictly within distance $\frac{4}{5}5^k$ of x^k, and therefore at distance greater than $5^k - \frac{4}{5}5^k = \frac{1}{5}5^k = 5^{k-1}$ from z^m. Now, since points at level $Y_{5^{k-1-\log_5 b}}$ have radius $\frac{5^{k-1}}{b}$, we may conclude that no descendants of x^k at level $Y_{5^{l-1-\log_5 b}}$ or lower can b-cover z^m. Therefore, only descendants of x^k in levels $Y_{5^{k-1}}$ down to $Y_{5^{k-1-\log_5 b}}$ can b-cover z^m, and there are only $b^{O(\lambda)}$ such descendants.

If x^k does not survive in the hierarchy, consider instead x^k's highest surviving descendant $\tilde{x}^l$ $(k > l)$, and repeat the previous argument for x^k on $\tilde{x}^l$: Since $\tilde{x}^l$ was not assigned as the step-parent of z^m, and z^m's step-parent or parent are above $\tilde{x}^l$, it must be that $\tilde{x}^l$ does not cover z^m. All descendants of $\tilde{x}^l$ are strictly within distance $\frac{4}{5}5^l$ of $\tilde{x}^l$, and therefore at distance greater than $5^l - \frac{4}{5}5^l = \frac{1}{5}5^l = 5^{l-1}$ from z^m. Now, since points at level $Y_{5^{l-1-\log_5 b}}$ have radius $\frac{5^{l-1}}{b}$, we may conclude that no descendants of $\tilde{x}^l$ at level $Y_{5^{l-1-\log_5 b}}$ or lower can b-cover z^m. Therefore, only descendants of $\tilde{x}^l$ in levels $Y_{5^{l-1}}$ down to $Y_{5^{l-1-\log_5 b}}$ can b-cover z^m, and there are only $b^{O(\lambda)}$ such descendants. $\square$

Proof of low degree

Now that we have the structural lemma, we can prove that the final spanner has degree $(1/\varepsilon)^{O(\lambda)}$. Recall that there are two types of edges incident on a point. Type I edges include parent-child and step-parent edges, and Type II edges include three different types of lateral edges. In proving low degree for the spanner of the final hierarchy, it will again prove useful to refer back to the spanner of the intermediate hierarchy.

Type I edges. For each occurrence of a point y in the intermediate hierarchy, y possesses $2^{O(\lambda)}$ parent-child edges. Similarly, for each occurrence of y, y possesses a single step-parent (and an edge to this step-parent). We will show that occurrence y^l can serve as a step-parent for at most $2^{O(\lambda)}$ other points which it covers.

To see that the occurrence serves as a step-parent for at most $2^{O(\lambda)}$ other points, note that each step-child of y^l has a unique ancestor in level Y_{5^l} of the full hierarchy; this ancestor did not survive in the intermediate hierarchy. By Property 2, the distance from this step-child to its ancestor is less than $\frac{4}{5}5^l$. The distance from step-child to y is less than 5^l (since y covers the step-child), so the distance from the ancestor to y is less than $\frac{4}{5}5^l + 5^l = \frac{9}{5}5^l$. There are $2^{O(\lambda)}$ points in level Y_{5^l} of the full hierarchy that are this close to y, so $y \in Y_{5^l}$ can have only $2^{O(\lambda)}$ step-children.

Since a point in the final hierarchy replaces at most three occurrences of points in the intermediate hierarchy, each point has at most $2^{O(\lambda)}$ Type I

edges incident upon it.

Type II edges. For each occurrence of point y in the hierarchy, say at level Y_{5^l}, y is given lateral edges to each point x^l of the full hierarchy that satisfies $d(y^l, x^l) \leq c \cdot 5^l$ and survives in the intermediate hierarchy. These account for $c^{O(\lambda)}$ edges incident on y.

If there exists a point x^l in the full hierarchy that satisfies $d(y^l, x^l) \leq c \cdot 5^l$ but does not survive in the intermediate hierarchy, then y is given a replacement lateral edge to x^l's highest surviving descendant (but only if the descendant's step-parent is above Y_{5^l}). This accounts for an additional $c^{O(\lambda)}$ edges incident on y.

Let w^j $(j > l)$ be the step-parent of y^l in the intermediate hierarchy, or its parent if it has no step-parent. If $j > l + 1$, then y^l possessed ancestors in levels $Y_{5^{j-1}}$ through $Y_{5^{l+1}}$ of the full hierarchy that did not survive in the intermediate hierarchy. For each such ancestor $\widetilde{y}^k$ $(j > k > l)$, y was given replacement lateral edges to all points of the full hierarchy within distance $c \cdot 5^k$ of $\widetilde{y}^k$ that survive in the intermediate hierarchy. Since y is $\frac{4}{5}$-covered by each ancestor, it is necessarily $(c + \frac{4}{5})$-covered by each point to which it is given a replacement lateral edge. By Lemma 4.2.3, there are only $c^{O(\lambda)}$ points in levels Y_{5^j} down to $Y_{5^{l+1}}$ that $(c + \frac{4}{5})$-cover y^l, so this accounts for only $c^{O(\lambda)}$ additional replacement lateral edges incident on y.

Now, recall that lateral refinement edges take two points connected by lateral and lateral replacement edges, and connect each point to every child of the other, as well as connecting the children of one point to the children

of the other. For each lateral or lateral replacement edge, this adds $2^{O(\lambda)}$ refinement edges incident on y, for a total of $c^{O(\lambda)}$ refinement edges.

It follows that a point occurrence accounts for $c^{O(\lambda)}$ lateral, lateral replacement, or lateral refinement edges. Since a point in the final hierarchy replaces at most three occurrences of points in the intermediate hierarchy, each point has at most $c^{O(\lambda)}$ Type II edges incident upon it. We may conclude:

Theorem 4.2.4. *The degree of the final spanner is* $c^{O(\lambda)} = (1/\varepsilon)^{O(\lambda)}$.

4.2.7 Dynamic updates

In this section we discuss how to maintain the spanner dynamically under insertions and deletions of points to the set. It suffices to show how to maintain T^1 and the intermediate spanner, since the final spanner is yielded by point replacement applied to the intermediate spanner, and it is straightforward to maintain the replacements as updates occur.

Maintenance of T^1

We begin with the dynamic maintenance of the spanning tree T^1. (Recall that T is maintained dynamically by the centroid path decomposition.) A single insertion into the point set translates into the insertion of $2^{O(\lambda)}$ nodes in T. These new nodes include only a single leaf node; this is the leaf node storing the newly inserted point. By construction, the new leaf node will appear in

T^1, although the newly added internal nodes each have at most one child and will not appear in T^1. It is however possible that the addition of the leaf node may result in a preexisting internal node of T being added to T^1; this can occur when the internal node had previously been compressed but now has the leaf node as a second child in T^1. (Note that this internal node may readily be found in $O(\log n)$ time using the centroid path decomposition). By construction, no other internal node may be added to T^1, although many internal nodes may have been added to T. (In particular, a node added to T as the result of a promotion is considered to have no real descendants at the time of promotion, and does not appear in T^1.)

Similarly, the deletion of a point results in a single leaf node of T being marked as deleted, and that leaf node being removed from T^1. The removal of a leaf node from T^1 may result in the removal (due to contraction) of a single internal node from T^1. (Again, this internal node may be found in $O(\log n)$ time using the centroid path decomposition).

It follows that T^1 – and the intermediate hierarchy it represents – can be maintained along with T in $O(\log n)$ update time. An update to the point set translates into at most two updates in the intermediate hierarchy.

Maintenance of the intermediate spanner

Now that we have detailed the changes that occur to T^1, we can show how to maintain the intermediate spanner. Before we begin, we will describe two query subroutines which we will make use of.

Colored ancestor queries. Consider a tree with some nodes that are *colored*. A *colored ancestor query* $\langle v,\mathbf{c}\rangle$ on node v and color $\mathbf{c}$ asks for the lowest ancestor of v which is colored $\mathbf{c}$. (Elsewhere this is called a *marked ancestor query* [1].) Multiple colored ancestors can be found by executing a series of queries, each subsequent query on the node returned by the previous query. Colored ancestor queries can be supported (for a constant number of colors) in $O(\log n)$ query and update time. We could accomplish this, for example, by making use of the dynamic centroid path decomposition of the tree. For each centroid path, the $\mathbf{c}$-colored nodes on that path are stored in a balanced tree. Then an ancestor query $\langle v,\mathbf{c}\rangle$ is executed by first discovering the centroid path on which v is found, and then finding the highest $\mathbf{c}$-colored node on that path, all in $O(1)$ time. If the highest $\mathbf{c}$-colored node on the path is above v, then a binary search on the balanced tree returns the lowest ancestor. If the highest $\mathbf{c}$-colored node on the path is below v (or there are no $\mathbf{c}$-colored nodes on the path), then the search ascends to the first ancestral centroid path that has $\mathbf{c}$-colored nodes (in $O(\log n)$ time), and returns the lowest $\mathbf{c}$-colored node on that path.

Highest surviving descendant queries. Consider the full and intermediate hierarchies. A *highest surviving descendant query* $\langle v\rangle$ provides a node v of T that does not survive in T^1, and asks for the highest descendant of v that survives in T^1. A highest surviving descendant query can be supported in $O(\log n)$ time using the centroid path decomposition. We first locate the centroid path of v, and then consult the centroid path's balanced tree for

real nodes (introduced in Section 3.6.4) in $O(\log n)$ time. If the balanced tree contains no nodes below v, then there is no surviving descendant. If it contains two or more nodes below v, then the highest one of these is the highest surviving descendant. If it contain a single node, then the search for a surviving descendant continues on the off-path subtree of this node. The new search takes only $O(1)$ time on the next centroid path (since the query node is at the top of the path and so a binary search on the balanced tree is not necessary), and if need be can descend all centroid paths in $O(\log n)$ time.

We can now proceed to describe the maintenance of the intermediate spanner. We will focus on Type I and Type II edges separately.

Type I edges. A newly added occurrence x^l in the intermediate hierarchy is given parent-child edges to its new parent and children, and these may be readily found by consulting tree T^1. If a child of x^l formerly had a different parent, the old parent-child edge is removed.

We must also locate the step-parent of x^l, which is the lowest point w^k ($k > l$) that covers x^l ($d(w^k, x^l) < 5^k$), if any, and is below x^l's parent. To find this point, we turn to the full hierarchy, and recall that the distance from x^l to its ancestor in level Y_{5^k} is less than $\frac{4}{5}5^k$, so the distance from w^k to this ancestor node is less than $5^k + \frac{4}{5}5^k = \frac{9}{5}5^k$. To find the point that covers x^l, we search the full hierarchy for the lowest ancestor of x^l that has a neighboring point within distance $\frac{9}{5}$ of its radius, and check if the neighbor covers x^l. If it does not, we find the next lowest ancestor of x^l with this property, stopping

when the parent level is reached. It follows from Lemma 4.2.3 that at most $2^{O(\lambda)}$ ancestor neighbors can be inspected. Such an ancestor search may be executed using T: We maintain a coloring scheme where each node is colored **r** if it stores a point that has a neighbor in the intermediate hierarchy within $\frac{9}{5}$ of its radius. We execute a colored ancestor query on the current ancestor of x^l to find the next lowest ancestor colored **r**. At most $2^{O(\lambda)}$ searches are undertaken, each requiring $O(\log n)$ time, so all this can be done in $2^{O(\lambda)} \log n$ time.

If x^l is the lowest point in T^1 covering some point z^m $(l > m)$, then x^l is z^m's step-parent. z^m must be given a step-parent edge to x, and the edge from z^m to its old step-parent edge (if any) is deleted.

To find all step-children of x^l, we note that the distance from a step-child z^m to x^l is at most 5^l, and by Property 2 the distance from x^l to the ancestor of z^m in level Y_{5^l} of the full hierarchy is less than $5^l + \frac{4}{5}5^l = \frac{9}{5}5^l$. For each of the $2^{O(\lambda)}$ points in level Y_{5^l} of the full hierarchy that is within distance $\frac{9}{5}5^l$ of x^l and does not survive in the hierarchy, we check if its highest surviving descendant is covered by x^l. If the descendant is covered by x^l, and x^l is lower than the point's current step-parent (if any), then x^l is its new step-parent and replaces the previous step-parent. This entails $2^{O(\lambda)}$ highest surviving descendant queries, and can be done in $2^{O(\lambda)} \log n$ time.

When a point occurrence is removed from the intermediate hierarchy, all edges to that point are deleted. Its parent and child are instead given parent-child edges to each other, and the child may gain a new step-parent. If the

removed point was a step-parent of some other point, that point is given a new step-parent; its new step-parent is located as above. This can all be done be done in $2^{O(\lambda)} \log n$ time.

Type II edges. For a newly added occurrence x^l in the intermediate hierarchy, x^l is given lateral edges to each point in the intermediate hierarchy at level Y_{5^l} within distance $c \cdot 5^l$ of x^l. There are $c^{O(\lambda)}$ such points, and they may be found in $c^{O(\lambda)}$ time by ascending and then descending $O(\log c)$ levels of the tree T.

If there is a point within distance $c \cdot 5^l$ of x^l in the full hierarchy, but that point does not survive in the full hierarchy, then its highest surviving descendant is given edges to x^l in the intermediate hierarchy. (This occurs only if the descendant's step-parent is above x^l.) There may be $c^{O(\lambda)}$ such points in level Y_{5^l} of the full hierarchy, and we can execute a highest surviving descendant on each one in $c^{O(\lambda)} \log n$ time.

Let w^j $(j > l)$ be the step-parent of x^l in the intermediate hierarchy, or its parent if it has no step-parent. If $j > l + 1$, then x^l possessed ancestors in levels $Y_{5^{j-1}}$ down to $Y_{5^{l+1}}$ of the full hierarchy that did not survive in the intermediate hierarchy. For each such ancestor $\widetilde{x}^k$ $(j > k > l)$, x^l must be given replacement lateral edges to all points of the full hierarchy within distance $c \cdot 5^k$ of $\widetilde{x}^k$ that survive in the intermediate hierarchy. We can use colored ancestor queries to locate each ancestor in levels $Y_{5^{j-1}}$ down to $Y_{5^{l+1}}$ that has surviving points within c times its radius (and given the ancestor we can find the nearby surviving points). To find these ancestors, we maintain a

coloring scheme where each a point y^l of the full hierarchy is colored **b** if it is within distance $c \cdot 5^l$ of some level Y_{5^l} point that survives in the intermediate hierarchy. We have already shown that each point has at most $c^{O(\lambda)}$ lateral replacement edges incident upon it, which implies that only $c^{O(\lambda)}$ ancestors must be located, and so this can all be done in $2^{O(\lambda)} \log n$ time.

If a newly added occurrence x^l is the new step-parent of z^m, then all lateral replacement edges between z^m and points above level Y_{5^l} are deleted. This can be done in $c^{O(\lambda)}$ time.

When a lateral or lateral replacement edge is deleted or inserted, $2^{O(\lambda)}$ lateral refinement edges are deleted or inserted as well. This has no asymptotic effect on the run time of lateral or lateral refinement edge deletion or insertion.

If x^l is deleted from the hierarchy, all lateral or lateral replacement edges due to x^l are deleted from the hierarchy. If x^l served as a step-parent of some point, that point is given a new step-parent and possibly new lateral edges. These can be found as above.

Lemma 4.2.5. *The intermediate spanner can be maintained in* $c^{O(\lambda)} \log n = O(\frac{\log n}{\varepsilon^{O(\lambda)}})$ *update time.*

Recall that the final spanner is derived from the intermediate spanner by replacing point occurrences in the intermediate hierarchy with their assigned points.

Corollary 1. *The final spanner can be maintained in* $O(\frac{\log n}{\varepsilon^{O(\lambda)}})$ *update time.*

Chapter 5

Further applications

We have already described the nearest neighbor search structure and its major application, the spanner. In this chapter we discuss two additional applications of the search structure. These are the maintenance of the closest pair of points in the point set, and the extraction of a well separated pairs decomposition of the point set.

5.1 Closest pair

An application of the spanner is dynamic maintenance of the closest pair of points in the set S. Note that in a $(2-\varepsilon)$-spanner ($\varepsilon > 0$), the pair (or pairs) of closest points must have an edge between them, or else their spanner stretch would be greater than $2-\varepsilon$. By storing the edges in a heap based on weight, we can answer a closest pair query in $O(1)$ time.

5.2 Well separated pairs decomposition

A *Well Separated Pairs Decomposition* of a point set X with constant $s > 0$ is a set of pairs $\{\{A_1, B_1\}, \ldots, \{A_l, B_l\}\}$ such that

- $A_i, B_i \in X$ for every i.
- $A_i \cap B_i = \emptyset$ for every i.
- $\cup_{i=1}^{l} A_i \otimes B_i = X \otimes X$.
- $d(A_i, B_i) \geq s \cdot \max\{\text{diam}(A_i), \text{diam}(B_i)\}$.

The WSPD was introduced by Callahan and Kosaraju [12], who gave a sequential algorithm for its derivation. They gave a dynamic data structure for X in d-dimensional Euclidean dimension that supports updates in $2^{O(d)} \log n$ time; given this data structure, a WSPD can be derived in $s^{O(d)}n$ time. Further, their WSPD has only $s^{O(d)}n$ pairs. For X in doubling dimension λ, [25] showed how to construct a linear size WSPD in $2^{O(\lambda)} n \log n + s^{O(\lambda)}$ time. Our navigating net data structure immediately yields a dynamic structure that supports updates in $2^{O(\lambda)} \log n$ time; given our navigating net, a WSPD can be found in $s^{O(\lambda)}n$ time.

Given our spanning tree T', the procedure for deriving a WSPD is a straightforward extension of the one given in [12]. For the purpose of the algorithm, we will treat all points of the hierarchy as if they were represented explicitly. Our algorithm is defined recursively, beginning at the top point

of the hierarchy: Let V be a set of points at level Y_{5^k} which have real descendants in S. Consider in turn each pair of points of V, say x and y. if $d(x, y) \geq 2s5^k$, then take x and y to be a well-separated pair. Otherwise, call the algorithm on the children of x and y in S.

As in [12], this algorithm produces a WSPD. As a consequence of the packing property applied to the navigating net, the algorithm runs in $s^{O(\lambda)}n$ time and produces $s^{O(\lambda)}n$ pairs.

Bibliography

[1] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. *Symposium on Foundations of Computer Science*, 534–544, 1998.

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

[3] S. Arya, D. M. Mount, and M. Smid. Dynamic algorithms for geometric spanners of small diameter: Randomized solutions. *Computational Geometry: Theory and Applications*, 13:91–107, 1999.

[4] A. Bagchi, A. L. Buchsbaum and M. T. Goodrich. Biased skip lists. *Algorithmica*, 42(1):31–48, 2005.

[5] R. E. Bellman. *Adaptive Control Processes*. Princeton University Press, 1961.

[6] S. W. Bent, D. D. Sleator and R. E. Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):545–68, 1985.

[7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[8] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *J. Alg.*, 1(4):301–358, 1980.

[9] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. *International Conference on Machine Learning*, 97–104, 2006.

[10] A. Borodin, R. Ostrovsky and Y. Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. *Algorithms and Combinatorics Series* 3143, 252–274, Springer-Verlag 1999.

[11] P. Bose, J. Gudmundsson, and P. Morin. Ordered theta graphs. *Computational Geometry: Theory and Applications*, 28:11–18, 2004.

[12] 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. ACM*, 42:67–90, 1995.

[13] E. Chavez, G. Navarro, R. Baeza-Yates, and J. L. Marroquin. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[14] T. Chan. Approximate nearest neighbor queries revisited. *Symposium on Computational Geometry*, 352–358, 1997.

[15] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete Computational Geometry*, 10:377–409, 1993.

[16] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Comput.*, 17:830–847, 1988.

[17] K. L. Clarkson. An algorithm for approximate closest-point queries. *Symposium on Computational Geometry*, 160–164, 1994.

[18] K. L. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.

[19] R. Cole and R. Hariharan. Dynamic lca queries. *SIAM J. on Comput.*, 34(4):894–923, 2005.

[20] J. Erickson. New lower bounds for convex hull problems in odd dimensions. *SIAM J. on Comput.*, 28(4):1–9, 1999.

[21] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *J. Alg.*, 24(1):37–65, 1997.

[22] J. Gao, L. Guibas, and A. Nguyen. Deformable spanners and applications. *Computational Geometry: Theory and Applications* 35(1):2–19, 2006.

[23] L. Gottlieb and L. Roditty. Improved algorithms for fully dynamic geometric spanners and geometric routing. *ACM Symposium on Discrete Algorithms*, 591–600, 2008.

[24] A. Gupta, R. Krauthgamer, and J. R. Lee. Bounded geometries, fractals, and low-distortion embeddings. *IEEE Symposium on Foundations of Computer Science*, 534–543, 2003.

[25] S. Har-Peled and M. Mendel. Fast construction of nets in low dimensional metrics, and their applications. *SIAM J. Comput.*, 35(5):1148–1184, 2006.

[26] D. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. *ACM Symposium on Theory of Computing*, 63–66, 2002.

[27] L. Roditty. Fully dynamic geometric spanners. *ACM Symposium on Computational Geometry*, 373–380, 2007.

[28] R. Krauthgamer and J. R. Lee. Navigating nets: simple algorithms for proximity search. *ACM-SIAM Symposium on Discrete Algorithms*, 798–807, 2004.

[29] R. Krauthgamer and J. R. Lee. The black-box complexity of nearest neighbor search. *Theoretical Computer Science*, 348(2–3):262–276, 2005.

[30] J. S. Salowe. Constructing multidimensional spanner graphs. *Int. J. Comput. Geometry Appl*, 1(2):99–107, 1991.

[31] J. Soares. Approximating euclidean distances by small degree graphs. *Discrete & Computational Geometry*, 11:213–233, 1994.

[32] P. M. Vaidya. A sparse graph almost as good as the complete graph on points in K dimensions. *Discrete & Computational Geometry*, 6:369–381, 1991.

[33] G. M. Ziegler. *Lectures on Polytopes*, Volume 152 of *Graduate Texts in Mathematics*. Springer-Verlag 1994.