Εθνικο Καποδιστριακο Πανεπιστημιο Αθηνων
Τμημα Μαθηματικων
Μεταπτυχιακο Προγραμμα Λογικης Και Αλγοριθμων

Exploiting the Structure of the Data in Approximate Nearest Neighbor Search

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αριστοτέλη-Εμμανουήλ Θάνου-Φίλη

Επιβλέπων: Ιωάννης Εμίρης
Καθηγητής Ε.Κ.Π.Α.

Αθήνα, Αύγουστος 2012

....
Copyright © Αριστοτέλης-Εμμανουήλ Θάνος-Φίλης, 2012.
Με επιφύλαξη παντός δικαιώματος. All rights reserved .

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Καποδιστριακού Πανεπιστημίου Αθηνών.

Abstract

Nearest neighbor searching (NNS) is a fundamental problem with several important applications. To accelerate the queries, we exploit the fact that data often exhibits highly nonrandom spatial patterns. We consider input points almost lying on about $\log^t n$ unknown lines in a space of constant dimension d, where n is the number of points and t constant. The lines are distributed uniformly in a bounding sphere, the points are distributed uniformly on each line, and their number per line varies from $\Omega(\log^t n)$ to $O(n)$. Queries take $O(\log\log n/\epsilon^{O(d^2)})$ expected time, which is exponentially faster than without structure, using optimal space $O(n)$, and return a $(1+\epsilon)$-approximate nearest neighbor, for any given $\epsilon > 0$. Ignoring the step of NNS on a line, queries take $O(\log^2 \log n/\epsilon^{O(d^2)})$ with high probability. Our key step is to employ a reduction of determining the nearest line to NNS among points.

Περίληψη

Η εύρεση του πλησιέστερου γείτονα είναι ένα καίριο πρόβλημα με πολλές σημαντικές εφαρμογές. Για να επιταχύνουμε το χρόνο εύρεσης του κοντινότερου γείτονα εκμεταλλευόμαστε το ότι πολλές φορές τα δεδομένα ακολουθούν μη τυχαία πρότυπα. Συγκεκριμένα, υποθέτουμε ότι τα σημεία που δεχόμαστε ως είσοδο βρίσκονται πάνω σε $\log^t n$ άγνωστες ευθείες σε κάποιο χώρο σταθερής διάστασης d, όπου n είναι ο αριθμός των σημείων και t είναι μια σταθερά. Οι ευθείες είναι κατανεμημένες ομοιόμορφα σε μια περιορισμένη μπάλα και τα σημεία είναι ομοιόμορφα κατανεμημένα πάνω σε κάθε ευθεία. Δεχόμαστε ότι κάθε ευθεία θα έχει από $\Omega(\log^t n)$ έως $O(n)$ σημεία. Η εύρεση ενός κατά προσέγγιση πλησιέστερου γείτονα χρειάζεται $O(\log\log n/\epsilon^{O(d^2)})$ αναμενόμενο χρόνο, ο οποίος είναι εκθετικά μικρότερος από το χρόνο που θα χρειαζόταν αν δεν εκμεταλλευόμασταν τη δομή των σημείων. Για κάθε δοσμένο $\epsilon > 0$ ο αλγόριθμος επιστρέφει ένα $(1+\epsilon)$-κατά προσέγγιση πλησιέστερο γείτονα, χρησιμοποιώντας βέλτιστο χώρο $O(n)$. Αν αγνοήσουμε την εύρεση του πλησιέστερου γείτονα ανάμεσα σε σημεία που βρίσκονται πάνω σε μία ευθεία, ο αλγόριθμος χρειάζεται χρόνο $O(\log^2\log n/\epsilon^{O(d^2)})$ με μεγάλη πιθανότητα. Το βασικό βήμα στον αλγόριθμο είναι η εφαρμογή μιάς αναγωγής από την εύρεση κοντινότερης ευθείας, στην εύρεση πλησιέστερου σημείου.

Acknowledgments

I would like to deeply thank my supervisor Prof. Emiris. I feel very lucky I had the chance to be taught by him as he has influenced me in a very positive way not only academically but also on a personal level.

Aristotelis-Emmanouil Thanos-Filis

Contents

1 Introduction **4**

2 Nearest Neighbor Search **6**

2.1 Structures for Approximate NNS 6

2.2 Balanced Box-Decomposition Trees 8

3 Exploiting the Structure of the Data **9**

3.1 Preprocessing, and Query Algorithm 9

3.2 Correctness 11

3.3 Time Complexity 12

4 Conclusion **17**

4.1 Conclusion and Future Work 17

Chapter 1

Introduction

In this thesis we consider the problem of approximate nearest neighbor search in the case where the input points are sampled from a polylogarithmic number of line segments in a bounding ball. The queries may be arbitrary in the ball. Assuming that the lines segments are chosen uniformly from the set of chords of the bounding ball and assuming further that the points are uniformly distributed on each segment, we achieve queries in expected $O(\log\log n/\epsilon^{O}(d^2))$ time and $O(\log^2\log n/\epsilon^{O}(d^2))$ time with high probability.

We use a reduction from [16], that maps lines and points in $\mathbb{R}^d$ to points in $\mathbb{R}^{d'}$, where $d' = \Theta(d^2)$, such that line-point distances in $\mathbb{R}^d$ are equal, up to a linear transformation, to point-point distances in $\mathbb{R}^{d'}$. This reduction enables the use of a dynamic data structure for NN search on points, so that, when answering a query for point q, the lines in L can be listed in order of their approximate distance to q. As each line l is listed, the NN to q in l is found, which gives an upper bound on the NN distance to q. When that the next line l to be listed is farther from q than that upper bound, the query algorithm stops. The distributional assumptions imply that $O(1)$ lines will be examined, in expectation, and the distributional assumptions on the points on each line imply that interpolation search can be used to find the NN of q on a given line l in $O(\log\log n)$ time.

The input model we consider is specialized but it is a first step to exploit directly the structure of the data points. A scenario in which this input model would fit well is the following: Some spaceships moving in straight lines in space are taking a photograph every 1 minute and for each photograph the position of the spaceship and the time is saved. Given a location-point in space we want to find a relatively recent photograph of the location which is the nearest possible to the point. Hence the data-points (i.e. the locations of the photographs taken) are on lines and almost equidistant on the lines since the spaceships have a constant speed. Moreover we can assume that the lines are distributed uniformly in space since we want the spaceships to cover as much locations as possble.

Our approach is original in that it improves the query performance without sacrificing space usage, by exploiting directly (on a straight forward way) the nonrandom pattern of

the dataset of points. Let P be a set of n points in $\mathbb{R}^d$. We assume they (approximately) lie on certain lines, but the lines are unknown. Given query point $q \in \mathbb{R}^d$ and approximation factor $\epsilon > 0$, we seek a point $v \in P$ such that $\text{dist}(q,v) \leq (1+\epsilon)\text{dist}(q,u)$, for all $u \in P$. We make the following hypotheses:

i. The number of lines on which the points lie is roughly $\log^t n$ where t is a positive integer constant.

ii. Each line contains at least $\log^t n$ points.

iii. Points are equidistant on each line, or picked independently and uniformly at random on each line (segment of the line inside the bounding sphere).

iv. The lines are picked independently and uniformly at random in a bounding sphere (in the sense of Remark 3.2).

Hypothesis (i), namely the points (almost) lying on a rather small number of lines, is required in order for the dataset to possess some significant structure. Hypothesis (ii) is quite loose and allows lines to contain very different numbers of points, even up to $O(n)$ points. Having the same number of lines and lower bound on the number of points per line is a technical assumption which can be removed, as discussed in Sect. 4.1. Hypotheses (iii) and (iv) are necessary for the complexity analysis; in their absence, our approach is still valid but with lower performance.

The input points may be generated as follows. A bounding sphere B is fixed and a number of lines that intersect with the sphere is picked uniformly at random (see Remark 3.2). For the segment of each line inside the sphere, a number of points is picked to approximately lie on this segment. The set of all points on these segments is the input dataset P. We consider the case that the points on each segment are picked uniformly at random, and the case that the points are equidistant on the segment. Note that we are not given the lines.

Our contribution is an approximate nearest neighbor algorithm, that achieves expected query time $O(c \log\log n)$, where $c \leq d'\lceil 1 + 6d'/\epsilon \rceil^{d'} = O(\epsilon^{-d'})$, and $d' = O(d^2)$, using optimal space $O(n)$. Clearly, $c = O(1)$ for a given ϵ, if d is constant; in the conclusion, we discuss general dimension. Furthermore, if we ignore searching among points on a line, a query time of $O(\log^2 \log n)$ is guaranteed with high probability.

To the best of our knowledge, this is the first algorithm that exploits directly the structure of the dataset of points. Of course a large number of data structures have been proposed for both approximate and exact search of pointsets that satisfy natural structural conditions such as having low doubling dimension [15],[14],[18], but in these works they take advantage of the structure of the space or the randomized way their structure is built.

The standard nearest neighbor search algorithms, e.g. based on BBD trees, would be exponentially slower, with query time logarithmic in n, assuming they used optimal space; the same holds for the more recent work on line queries [2].

Chapter 2

Nearest Neighbor Search

Nearest neighbor searching (NNS), also known as similarity searching, is a fundamental question with several important applications, including machine learning, geometric inference, and high-dimensional optimization. A lot of current research focuses on this problem, e.g. [2, 21, 17, 12], with impressive, and often optimal, results. One of the main remaining open questions, and a means to further improve the query complexity, is to exploit the fact that, in practical situations, e.g. bioinformatics and image analysis, data (or queries) exhibit high correlation, such as temporal and spatial locality. We focus on approximate NNS when the dataset of points almost lie on few, unknown lines. This is the first step in investigating adaptive approximate NNS. Our algorithm achieves optimal space usage and exponential query speedup since query time is logarithmic in the number of lines instead of the number of points.

Let P be a set of n points in $\mathbb{R}^d$ and let $\text{dist}(p, p')$ be the *Euclidean distance* between any points p, p'. The nearest neighbor problem consists in reporting, given a query point q, its nearest neighbor $p \in P$ such that $\text{dist}(p, q) \leq \text{dist}(p', q)$, for all $p' \in P$. For this purpose, one preprocesses P into an appropriate data structure, called NN-structure. Since an exact solution to high-dimensional NNS requires heavy resources, research has focused on *approximate* nearest neighbors. Given parameter $\epsilon > 0$, a $(1 + \epsilon)$-approximate nearest neighbor (ϵ-NN) to a query q is any point $p \in P$ such that $\text{dist}(q, p) \leq (1 + \epsilon) \cdot \text{dist}(q, p')$, where p' is a nearest neighbor to q.

2.1 Structures for Approximate NNS

A classic data structure is k-d trees [9]. Each level of the tree represents a partition of space by an axis-perpendicular hyperplane. An interesting implementation is `FLANN` (Fast Library for Approximate Nearest Neighbor), which contains (probabilistic) algorithms and data-structures among which it chooses the most appropriate for the input [12].

Balanced Box-Decomposition (BBD) trees [23] offer query time $O(c \log n)$, $c \leq d\lceil 1 +$

$6d/\epsilon]^d$, using space $O(dn)$ where n is the number of points in the structure. BBD-trees allow the deletion of a point from the structure in $O(\log n)$ and are thus employed by our algorithm. They led to `ANN`, a state-of-the-art approximate nearest-neighbors software.

Approximate Voronoi Diagrams (AVD) are another relevant data structure. They offer a tradeoff between space complexity and query time based on parameter $\gamma : 2 \leq \gamma \leq \frac{1}{\epsilon}$ [21]. Then, the query takes $O\left(\log(n\gamma) + 1/(\epsilon\gamma)^{\frac{d-1}{2}}\right)$ and space is $O\left(n\gamma^{d-1}\log\frac{1}{\epsilon}\right)$. They are implemented on a hierarchical quadtree-based subdivision of space into cells, each storing a number of representative points, such that for any query point lying in the cell, at least one of the representatives is an approximate nearest neighbor.

A major approach for high dimensions is Locality Sensitive Hashing. In [1] they present an algorithm that almost matches the known lower bound and achieves query time $O(dn^{\epsilon^{-2}+o(1)})$, using $O(dn + n^{1+\epsilon^{-2}+o(1)})$ space. However, it is not clear that a "dual" of this approach might work in our setting.

An interesting generalization of the problem arises if we replace the pointset with a set of objects O, but there are only few results known. In $\mathbb{R}^3$, when O comprises disjoint polyhedra [24] presented a data structure of near quadratic size that answers an ϵ-NN query in $O(\log(n/\epsilon))$. In [25], they answer ϵ-NN queries when O is a set of triangles, segments, and points in convex position, in $O(\log^2 n/\epsilon^2)$ time using $O(n/\epsilon^2)$ space. In [6], they developed a data structure for ϵ-NN queries over a set of parallel segments. In high dimensions, [4] offers an algorithm for a set of k-flats with query $(d + \log n + 1/\epsilon)^{O(1)}$ but super-polynomial space in $2^{(\log n)^{O(1)}}$.

More recently, there have been results where the queries are lines for a dataset of points [2], or the dataset is a set of linear or affine subspaces with point queries [16], or both the dataset and the queries are linear or affine subspaces [17]. In [2] they present an algorithm for high dimensions which, for approximation $1+\epsilon$, achieves query time $O(d^3 n^{0.5+c})$, for arbitrary small $c > 0$, and space $O(d^2 n^{O(1/\epsilon^2+1/c^2)})$. We analyze and apply the results of [16] in the sequel, since they reduce finding the nearest line to the approximate NNS among points in a higher dimension. They alleviate the problem of high dimension experimentally but without guarantees; we shall address this issue in Sect. 4.1.

In [22, 20] it was shown how to answer efficiently planar point location queries with temporal locality, in optimal expected-case query time. In [7], they gave for the same problem spatially adaptive methods. For more than 2 dimensions, there are very few results in exploiting structure. The most relevant is [10], where they showed how to solve the approximate NNS in a distance-sensitive manner with data structures using space filling curves.

In [19] they study the Approximate Nearest Neighbor problem for metric spaces where the query points are constrained to lie on a subspace of low doubling dimension.

2.2 Balanced Box-Decomposition Trees

In this section we introduce the balanced box-decomposition tree or BBD-tree [23], which is the primary data structure used in our algorithm. It is among the general class of geometric data structures based on a hierarchical decomposition of space into d- dimensional rectangles whose sides are orthogonal to the coordinate axes. The principal difference between the BBD-tree and the other data structures listed above is that each node of the BBD-tree is associated not simply with a d-dimensional rectangle, but generally with the set theoretic difference of two such rectangles, one enclosed within the other.

It is constructed through the repeated application of two operations, fair splits (or simply splits) and shrinks. A fair split partitions a cell by an axis-orthogonal hyperplane. The two children are called the low child and high child, depending on whether the coordinates along the splitting coordinate are less than or greater than the coordinate of the splitting plane. A shrink partitions a cell into disjoint subcells, but uses a box (called the shrinking box) rather than a hyperplane to do the splitting. It partitions a cell into two children, one lying inside (the inner child) and one lying outside (the outer child). The recursive construction algorithm is given a cell and a subset of data points associated with this cell. Each stage of the algorithm determines how to subdivide the current cell, either through splitting or shrinking, and then partitions the points among the child nodes. This is repeated until the number of associated points is at most one. A simple strategy is that splits and shrinks are applied alternately. This will imply that both the geometric size and the number of points associated with each node will decrease exponentially as we descend a constant number of levels in the tree.

An intuitive overview of the approximate nearest neighbor query algorithm follows. Given the query point q, we begin by locating the leaf cell containing the query point in $O(\log n)$ time by a simple descent through the tree. Next, we begin enumerating the leaf cells in increasing order of distance from the query point. We call this priority search. When a cell is visited, the distance from q to the point associated with this cell is computed. We keep track of the closest point seen so far. Let p denote the closest point seen so far. As soon as the distance from q to the current leaf cell exceeds $\text{dist}(q, p)/(1+\epsilon)$, it follows that the search can be terminated, and p can be reported as an approximate nearest neighbor to q. The reason is that any point located in a subsequently visited cell cannot be close enough to q to violate p's claim to be an approximate nearest neighbor.

BBD trees offer query time $O(c \log n)$, $c \leq d\lceil 1 + 6d/\epsilon \rceil^d$, using space $O(dn)$ where n is the number of points in the structure.

Chapter 3

Exploiting the Structure of the Data

3.1 Preprocessing, and Query Algorithm

In this section we describe our algorithm. First, it finds the lines on which the points lie. When a query point q is given, it finds the k nearest lines to q, each containing a pointset S_i, $i = 1, \ldots, k$. For each line, the algorithm finds the nearest point $u_i \in S_i$ to q, and returns the nearest point among $\{u_1, \ldots, u_k\}$. The algorithm's steps are:
Preprocessing steps:

(P_1) Find the lines on which the points lie and for each line store its points in sorted order.

(P_2) Map each line to a point in $\mathbb{R}^{d'}$, $d' > d$, and construct a dynamic NN-structure for these points.

Query steps:

(Q_1) Given a query point $q \in \mathbb{R}^d$, use the map in step P_2 to map it to the same space $\mathbb{R}^{d'}$, then use the NN-structure to find its closest point.

(Q_2) Remove the point found in Q_1 from the NN-structure.

(Q_3) Find the line in the original space that corresponds to the point found in Q_1.

(Q_4) Among the points (approximately) on this line, find point u_i nearest to q, and compute their distance, denoted by p.

(Q_5) Repeat steps Q_1 to Q_4:

1. If the new distance is less than p, update the value of p with the new distance.

2. If the line found is at distance $> p$, return the u_i that corresponds to p.

In step P_1 we compute the set of lines l_i: Iteratively pick a pair of points and check how many of the other points (approximately) lie on the line defined by the pair. We keep the lines that contain at least $\log^t n$ points. If there exist more than $\log^t n$ such lines, keep the $\log^t n$ with most points. The algorithm constructs two data structures for the pointset S_i on every line l_i: a binary search tree and a sorted array. In step P_2, we map the lines found in step P_1 to a set of points in $\mathbb{R}^{d'}$, where $d' = O(d^2)$, using the mapping in [16]. A dynamic NN-structure, namely BBD-tree, is built on all image points. We use this mapping to determine nearest lines to the query point.

After we are given a query point $q \in \mathbb{R}^d$, in step Q_1 the algorithm first maps it to a point $q' \in \mathbb{R}^{d'}$ and then queries the NN-structure to find an $(1 + \epsilon)$-approximate nearest neighbor s_1 of q'.

In step Q_2, we remove s_1 from the NN-structure. As a result, when we later search for the nearest point, the next approximate nearest point is returned. Point s_1, found in step Q_2, corresponds without loss of generality to l_1, which is the (almost) nearest line to q, returned by step Q_3. It satisfies the approximation factor as proven in Corollary 3.1. We compute the projection (foot) v_1 of q on l_1.

In step Q_4, we employ interpolation search [11, 3] to determine point $u_1 \in S_1$ which is closest to v_1. We compute the distance between q and u_1. Interpolation search, when the points follow the uniform distribution, has expected runtime $O(\log \log m)$, where m is the number of the points in the sorted array. In our case $m = O(n)$ because the line may contain as many points. Unfortunately, in the worst case the time is linear. Thus, we perform binary search if interpolation search has not terminated in time $O(\log n)$.

In spite of l_1 being almost nearest to q, this is not necessarily the case for u_1, because the points on l_1 may be relatively far from v_1. Thus, we use the NN-structure k times to find k approximate nearest lines and, for every line, we repeat the above procedure. Let ρ_j be the distance between q and l_j returned at the j-th iteration. Let point $u_j \in l_j$ be closest to q. The algorithm stops when:

$$\min_{j=1}^{k} \{\text{dist}(q, u_j)\} \leq \rho_k. \tag{3.1}$$

In Corollary 3.3 we determine k with high probability. To guarantee correctness, we do not fix k a priori but, instead, we compute nearest lines until bound (3.1) holds; this may increase the worst-case query time.

Mapping. We employ the mapping from [16]. We represent a line l_i by a $(d+1) \times (d-1)$ matrix Z whose first d rows contain orthonormal columns, representing the hyperplane orthogonal to l_i, and the last row contains the offset vector v in this hyperplane. We represent the query point q by $\hat{q} = (q^T, 1)^T$. For a symmetric $d \times d$ matrix A we define a vector $h(A)$ containing the entries of the upper triangular portion of A, with the diagonal

entries scaled by $1/\sqrt{2}$. Let $I = \text{diag}\{1, \ldots, 1, 0\}$ be a $(d+1) \times (d+1)$ matrix. The following transformations define maps

$$l_i \mapsto \hat{u}_i \in \mathbb{R}^{d'}, \quad q \mapsto \hat{v} \in \mathbb{R}^{d'}, \tag{3.2}$$

to points in $\mathbb{R}^{d'}$, where $d' = \frac{(d+1)(d+2)}{2} + 1 = O(d^2)$, where

$$\hat{u}_i = -\left(h(ZZ^T) + \frac{d-1}{d}h(I), c(l_i)\right), \quad \hat{v} = \gamma\left(\frac{1}{\|q\|^2}h(\hat{q}\hat{q}^T) + \frac{1}{d}h(I), 0\right),$$

with

$$c(l_i) = \sqrt{\frac{M^4 - \|ZZ^T\|_F^2}{2}} \quad \text{and} \quad \gamma = \sqrt{\frac{dM^4 - (d-1)^2}{d-1}}.$$

Here $\|\cdot\|_F$ stands for the Frobenius norm defined by the following trace and given by the formula: $\|ZZ^T\|_F^2 = Tr(ZZ^T(ZZ^T)^T) = d - 1 + 3\|v\|^2$, where v is the offset; $M > 0$ is a sufficiently large constant such that all $c(l_i) \in \mathbb{R}$ (thus it is determined by the line with the largest norm of the offset vector).

Proposition 3.1. *[16] Assume that σ is a linear or affine subspace, mapped to point $\hat{u}_i$, and q is mapped to $\hat{v}$. For μ, ν constants, it holds:*

$$dist^2(\hat{u}_i, \hat{v}) = \mu \cdot dist^2(\sigma, q) + \nu.$$

Let δ be the intrinsic dimension of σ. If σ is linear (i.e. contains the origin), which is not our case,

$$\mu = \frac{1}{\|q\|^2}\sqrt{\frac{\delta(d-\delta)}{d-1}}, \quad \nu = \left(1 - \frac{\delta}{d}\right)\left(\delta - \sqrt{\frac{\delta(d-\delta)}{d-1}}\right).$$

If $\delta = 1$, then $\nu = 0$, as in our case.

Thus, transformation (3.2) satisfies:

$$\text{dist}^2(\hat{u}_i, \hat{v}) = \mu \cdot \text{dist}^2(l_i, q), \tag{3.3}$$

3.2 Correctness

This section proves the correctness of our method.

Corollary 3.1. *Let q be the query point and l_i be a line and assume that after applying the mapping, y corresponds to q and x_i corresponds to l_i. Let OPT be the distance between y and its exact nearest neighbor and OPT_s be the distance between the query and its exact nearest line in $\mathbb{R}^d$. If the NN-structure returns x_i, then $dist(l_i, q) \leq (1+\epsilon)OPT_s$.*

Proof. It holds from (3.3) that $OPT^2 = \mu \cdot OPT_s^2$. If the NN-structure returns x_i for which $\text{dist}(x_i, y) \leq (1+\epsilon)OPT$, then:

$$\begin{aligned} \text{dist}^2(x_i, y) = \mu \cdot \text{dist}^2(l_i, q) &\Leftrightarrow (1+\epsilon)^2 OPT^2 \geq \mu \cdot \text{dist}^2(l_i, q) \Leftrightarrow \\ (1+\epsilon)^2 \mu \cdot OPT_s^2 \geq \mu \cdot \text{dist}^2(l_i, q) &\Leftrightarrow (1+\epsilon) OPT_s \geq \text{dist}(l_i, q). \end{aligned}$$

□

Theorem 3.1. *The above algorithm returns a $(1+\epsilon)$-approximate nearest neighbor.*

Proof. By Corollary 3.1, the algorithm starts with the $(1+\epsilon)$-approximate nearest line and, among its points, finds the one closest to the query q; it stores their distance p. It then finds the next nearest line, provided it is at distance less than p, finds the closest point on it, and may decrease the value of p accordingly. This continues until the next nearest line is at distance $> p$, which implies its points are at distance $> p$. Let l_i and l_j stand for the lines examined and not examined, respectively, by the algorithm. It follows that

$$(1+\epsilon) \min_j \text{dist}(l_j, q) \geq \min_i \text{dist}(l_i, q) > p.$$

For any point $p_j \in l_j$, for any j, it follows $\text{dist}(p_j, q) > p/(1+\epsilon)$. The algorithm returns p, and the data point realizing this distance to q, hence the claim is established.

□

If we fix k a priori and look for the k nearest neighbors, there is a small probability that the algorithm gives an incorrect answer, since the lines are chosen uniformly at random in the bounding sphere (Remark 3.2), hence they may not help us exploit the position of the points to accelerate queries. For this reason, we do not fix k but instead we find nearest lines until expression (3.1) holds. In this way we increase the worst-case query time but guarantee correctness.

3.3 Time Complexity

The running time of our algorithm depends on the NN structure we choose. If we assume that the dimension d is small in comparison to the number of points n, the best data structures are the BBD-trees and AVD. The advantage of the BBD-trees is that we can remove a point from the structure. Using this feature of the BBD-trees, we will be able to remove the point that corresponds to the approximate nearest line after we find it.

In the preprocessing step, finding the set of lines costs $\binom{n}{2} \cdot O(n) = O(n^3)$ time, which dominates the preprocessing time complexity. The sorting of points on every line costs $O(n \log n)$ time per line, for a total time in $O(n \log^{t+1} n)$. Storing takes a total of $O(n)$ space. The construction of the BBD-tree costs $O(d^2 \log^t n \log\log n)$ time and $O(d^2 \log^t n)$ space.

Query analysis. In the processing step we query the BBD-trees structure k times, so the time cost will be $O(kc \log\log n)$ where $c \leq d' \lceil 1 + 6d'/\epsilon \rceil^{d'}$ since the number of points in the structure is $\log^t n$. We perform an interpolation search on each of the k approximate nearest lines, using each line's sorted array. Since a line contains $\geq \log^t n$ points and there are $\log^t n$ lines, a line contains $\leq n - \log^{2t} n$ points. In the case where the points are equidistant on the lines, interpolation search returns the nearest point in O(1) time. In the case they are picked uniformly at random on the lines, interpolation search returns the nearest point in expected $O(\log\log n)$ time but, in the worst case, in $O(n)$ time. To deal with this we perform interpolation search for $O(\log n)$ time and, if the nearest point is not found, we use binary search in $O(\log n)$ time, using the line's binary search tree.

Remark 3.1. *We denote by F the time of finding the nearest point to q within the points of a line. The overall query time will be $O\left(k\left(c \log\log n + F\right)\right)$. From the above discussion, in the expected case, $F = O(\log\log n)$, whereas in the worst case, $F = O(\log n)$.*

Lemma 3.2 shows that the expected value $E[k] \leq 1$ and, with high probability, $k = O(\log\log n)$. In the worst case, our algorithm searches all lines, and the distribution of points on lines shall not allow the interpolation search to terminate in $O(\log\log n)$ time, which implies $k = O(\log^t n)$ and $F = O(\log n)$. Thus, the worst-case overall query time is $O(\log^{t+1} n)$.

Definition 3.1. *We define a* hyperannulus *with width r to be the set theoretic difference of two hyperspheres cocentred with radii R_1, R_2 such that $R_2 - R_1 = r$. We will say that a line* lies inside *a hyperannulus if the line intersects the hyperannulus but does not intersect the inner hypersphere.*

Remark 3.2. *A line $l_i \subset \mathbb{R}^d$ is defined by a direction unit vector $e_2 \in S^{d-2}$, where S^i is the i-dimensional unit sphere, and a point $b \in \mathbb{R}^d$ on the line. The latter point can be uniquely determined by a unit vector $e_1 \in S^{d-1}$ and a distance α from the origin. We pick l_i in the bounding sphere of radius R as follows: Without loss of generality, assume that the center of the sphere is the origin. Pick uniformly at random a unit vector $e_1 \in \mathbb{R}^d$, rooted at the origin. Then, pick uniformly at random a number α from the interval $[0, R]$, and let b be the point on the line defined by e_1 at distance α from the origin. Finally, pick uniformly at random a unit vector $e_2 \in \mathbb{R}^{d-1}$, lying on the orthogonal hyperplane to e_1, and rooted at b. The line l_i is defined by e_2.*

Lemma 3.1. *The probability for a line, which is picked uniformly at random as in Remark 3.2, to lie inside a hyperannulus depends only on its width and not on the radii of the spheres which define it, provided the hyperannulus is contained in the bounding sphere.*

Proof. A hyperannulus defined by two hyperspheres contains the same number of lines irrespective of their common center. Using Remark 3.2, a line lies inside the hyperannulus defined by hyperspheres with radii R_1, R_2 iff its distance α from their common center is in $[R_1, R_2]$. This is a random event that depends only on the choice of α, hence the

probability that a line lies inside the hyperannulus depends only on the hyperannulus' width.

□

Corollary 3.2. *If a hyperannulus of width r is contained in a bounding sphere of radius R and a line is picked uniformly at random inside the bounding sphere as in Remark 3.2, the probability that the line lies inside the hyperannulus is r/R.*

Proof. Without loss of generality, we assume that the bounding sphere is partitioned by cocentred hyperannuli $C_1, \dots, C_{R/r}$, all of width r. If a line is picked inside C_i, then this line does not lie inside any other C_j, $j \neq i$. We denote by U_i the event that a line lies inside the hyperannulus C_i. By the previous argument, we have $U_i \bigcap U_j = \emptyset$, $\forall i \neq j$. Let $P(U)$ denote the probability of event U. Using union bound we have:

$$1 = P\left(\bigcup_{i=1}^{R/r} U_i\right) = \sum_{i=1}^{R/r} P(U_i).$$

The probability for a line to lie inside any hyperannulus is fixed and given by Lemma 3.1. Then, $P(U_i) = P(U_j)$, $\forall i, j$. As a result, $P(U_i) = r/R$.

Lemma 3.1 implies that a hyperannulus of width r, inside a bounding sphere of radius R, contains a line with the same probability as every C_i. Hence, the probability for a line, which is picked uniformly at random in the sphere, to lie inside this hyperannulus is r/R.

□

Lemma 3.2. *The expected number k of approximate nearest lines that the algorithm finds, before the termination condition (3.1) is satisfied, is $E[k] \leq 1$.*

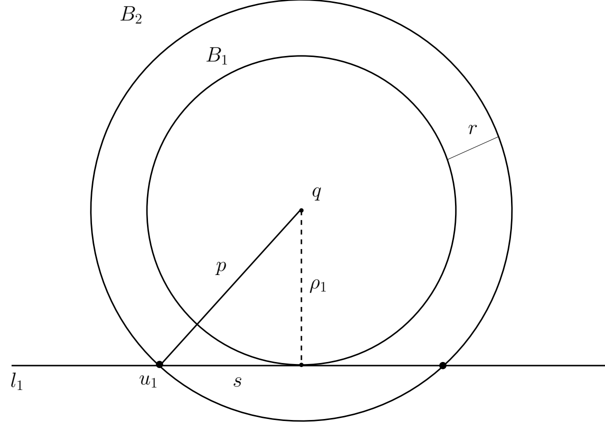
Proof. After the algorithm finds the first approximate nearest line l_1, we know that there does not exist a line inside the hypersphere B_1 centered at the query point q with radius $\frac{\rho_1}{1+\epsilon}$. If such a line existed, the approximation factor of the NNS would have been violated by Corollary 3.1. Let s denote the distance between u_1 and the projection of q on l_1, and let B_2 be the hypersphere centered at q with radius p (Fig. 3.1). We show that the expected number of lines inside this annulus is ≤ 1.

First suppose the hyperannulus is entirely contained in the bounding sphere. Every line contains $\geq \log^t n$ points and, in both cases of hypothesis (iii), the (expected) distance of two points is $\leq 2R/\log^t n$, where R is the radius of the bounding sphere. Then $s \leq R/\log^t n$. Let r denote the width of the hyperannulus. It holds that $p = \rho_1 + r$, and $p^2 = s^2 + \rho_1^2$ (Fig. 3.1), thus $s^2 = r^2 + 2r\rho_1 \Rightarrow r \leq s \Rightarrow r \leq R/\log^t n$. Applying Corollary 3.2, the probability of a line to be inside this hyperannulus is $1/\log^t n$. Since we pick independently $\log^t n$ lines, the expected number of lines in the hyperannulus is 1.

There may appear instances such that q, l_1 and u_1 define a hyperannulus that partly lies outside the bounding sphere, namely when q is closer to the surface of the bounding

sphere than to l_1. Since lines only exist inside the bounding sphere, k is the number of lines inside the intersection of the hyperannulus with this sphere. The expected number of lines inside this part of the hyperannulus is less than the expected number of lines in the entire hyperannulus.

□



Sq'hma 3.1: l_1 is the nearest line to the query q, and u_1 is the nearest point on the line to q. The next nearest line l_2 cannot be nearer to q than l_1 and hence cannot intersect with B_1. The algorithm will terminate if l_2 is at distance $> p$ to q, i.e. l_2 does not intersect B_2. The number k of iterations is bounded by the number of lines inside the hyperannulus defined by B_1, B_2.

We proved, in Sect. 3.3, that our algorithm achieves query time $O\left(k\left(c \log\log n + F\right)\right)$. Using Lemma 3.2 and that interpolation search has expected time complexity $O(\log\log n)$ (see Remark 3.1), we have the following:

Theorem 3.2. *Our algorithm returns an $(1+\epsilon)$-approximate nearest neighbor in expected query time $O(c \cdot \log\log n)$, using space $O(n)$, where $c \leq d'\lceil 1 + 6d'/\epsilon\rceil^{d'} = O(\epsilon^{-d'})$, $d' = O(d^2)$, and the space dimension d is constant.*

We now apply the theory of *Balls and bins* [26, Chap.5] to control the probability that the above complexity indeed occurs. Consider the process of tossing n balls into n bins. The tosses are made uniformly at random and independent of each other, which implies that the probability that a ball falls into any given bin is $1/n$.

Proposition 3.2. *With high probability, namely $\geq 1 - 1/n$, all bins have at most $3 \ln n / \ln\ln n$ balls.*

Corollary 3.3. *Let k be the number of approximate nearest lines the algorithm finds before the termination condition is satisfied. Then $k = O(\log\log n / \log\log\log n)$ with probability $\geq 1 - 1/\log^t n$.*

Proof. We use a simple reduction from the bins and balls problem. We saw before that the width r of the hyperannulus with inner hypersphere C_1 and outer hypersphere C_2 is at most $R/\log^t n$ (Fig. 3.1). We assume that the bounding sphere is partitioned with cocentred annuli with width $R/\log^t n$. The number of annuli needed to cover the bounding sphere is $\log^t n$, and any line picked lies inside a unique annulus. We consider these annuli to be bins and the $\log^t n$ lines, that are picked uniformly at random and independent of each other, are balls. We apply Proposition 3.2 to obtain that any annulus with width $R/\log^t n$ contains at most $3 \log t \log n / \log\log t \log n$ lines with probability $1 - 1/\log^t n$. □

It should be clear that the number of lines k found by the NN-structure is bounded by the number of lines in the hyperannulus formed by the query point q, its first approximate nearest line l_1 and its nearest point $u_1 \in l_i$ (Fig. 3.1). In case the hyperannulus lies partly outside the bounding sphere, k will be the number of lines in the intersection of the hyperannulus with the sphere. Thus, Corollary 3.3 holds in this case too.

Corollary 3.3 guarantees our algorithm achieves query time $O\left(\log\log n \left(c \log\log n + F\right)\right)$ with high probability. By combining it with Theorem 3.2, and Remark 3.1 for specifying F, we obtain our Main Theorem:

Theorem 3.3. *Suppose the points lie on some unknown lines in $\mathbb{R}^d$, $d = O(1)$, satisfying hypotheses (i) to (iv), and the queries are points. Our algorithm (preprocessing steps P_1, P_2, and processing steps Q_1 to Q_4) uses optimal space $O(n)$ to return an $(1+\epsilon)$-approximate nearest neighbor in query time $O\left(k\left(c \log\log n + F\right)\right)$, where k is the number of examined lines, F the cost of searching on a line, and $c = O(\epsilon^{-O(d^2)})$, which is constant for $\epsilon = O(1)$. The query time is bounded by $O(c \log\log n)$ in the expected case, $O(\log^{t+1} n)$ in the worst case, and $O\left(\log\log n \left(c \log\log n + F\right)\right)$, with high probability.*

Chapter 4

Conclusion

4.1 Conclusion and Future Work

This is a first effort to exploit structure of the data points, lying on roughly $\log^t n$ unknown lines, to achieve a query logarithmic in their number, i.e. exponentially faster than standard NNS, while using optimal space. In the worst case, when interpolation search fails to terminate in $o(\log n)$, or termination condition (3.1) requires that the algorithm examines all lines, our query becomes $O(\log^{t+1} n)$, comparable to standard NNS. Essentially our algorithm solves the problem even if points only approximately lie on lines.

We assumed the number of points per line is $\geq \log^t n$ (same as the number of lines) to apply Proposition 3.2 and bound k. We can generalize to lines with $\geq \log n$ points and employ the bounds for different (larger) number of balls than bins [13]. Clearly, the number of lines L determines query time, which is $O(k \log L + kF)$, since the more lines there are, the less structured we have.

For general dimension, we employ dimension reduction by random projection [8] of the points (obtained by the mapping) to $\mathbb{R}^\delta$, $\delta = O(\log n/\epsilon^2)$. This distorts distances in $\mathbb{R}^{d'}$ by a factor of $\leq 1 + \epsilon$, with high probability, assuming $d' \gg \log n$. Let the nearest neighbor of the mapped query q' be π_1 and the point actually found be π_2, then:

$$\text{dist}(q', \pi_2) \leq \frac{(1+\epsilon)^2}{1-\epsilon} \,\text{dist}(q', \pi_1).$$

It suffices for our algorithm to change the termination condition (3.1) to:

$$\min_{j=1}^{k} \{\text{dist}(q, u_j)\} \leq \frac{1+\epsilon}{1-\epsilon}\, \rho_k.$$

We choose the data structure in [5] to obtain query time $O\left(\delta^2 (\log m + 1/\epsilon)^{O(1)}\right)$ and $O(\delta^2 m^{O(1/\epsilon^2)})$ space, where $m = \log^t n$.

This is the first step in investigating adaptive approximate NNS; we plan to study the problem with points lying on objects other than lines, such as circles.

References

[1] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proc. FOCS*, pages 459–468, 2006.

[2] Alexandr Andoni, Piotr Indyk, Robert Krauthgamer, and Huy L. Nguyen. Approximate line nearest neighbor in high dimensions. In *Proc. SODA*, pages 293–301, 2009.

[3] Alexis C. Kaporis, Christos Makris, Spyros Sioutas, Athanasios K. Tsakalidis, Kostas Tsichlas, and Christos D. Zaroliagis. Dynamic interpolation search revisited. In *Proc. ICALP*, pages 382–394, 2006.

[4] Avner Magen. Dimensionality reductions in l_2 that preserve volumes and distance to affine spaces. *Discrete & Computational Geometry*, 38(1):139–153, 2007.

[5] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *SIAM J. Comput.*, 30(2):457–474, 2000.

[6] Ioannis Z. Emiris, Theocharis Malamatos, and Elias P. Tsigaridas. Approximate nearest neighbor queries among parallel segments. In *26th Europ. Workshop Comp. Geom. (EuroCG)*, pages 141–144, Dortmund,, 2010.

[7] John Iacono and Stefan Langerman. Proximate planar point location. In *Proc. Symp. Comp. Geom.*, pages 220–226, 2003.

[8] William Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conf. Modern anal. & prob. (1982)*, volume 26 of *Contemporary Mathematics*, pages 189–206. AMS, 1984.

[9] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[10] Jonathan Derryberry, Don Sheehy, Maverick Woo, and Danny Dominic Sleator. Achieving spatial adaptivity while finding approximate nearest neighbors. In *Proc. Canadian Conf. Comp. Geom.*, 2008.

[11] Kurt Mehlhorn and Athanasios K. Tsakalidis. Dynamic interpolation search. *J. ACM*, 40(3):621–634, 1993.

[12] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *Proc. VISAPP: Intern. Conf. Computer Vision Theory & Appl.*, pages 331–340, 2009.

[13] Martin Raab and Angelika Steger. Balls into bins: A simple and tight analysis. In *Proc. RANDOM*, pages 159–170, 1998.

[14] Piotr Indyk and Assaf Naor. Nearest-neighbor-preserving embeddings. *ACM Transactions on Algorithms*, 3(3), 2007.

[15] Robert Krauthgamer and James R. Lee. The black-box complexity of nearest neighbor search. In *ICALP*, pages 858–869, 2004.

[16] Ronen Basri, Tal Hassner, and Lihi Zelnik-Manor. Approximate nearest subspace search with applications to pattern recognition. In *Proc. Comp. Vision Pattern Recogn. (CVPR)*, 2007.

[17] Ronen Basri, Tal Hassner, and Lihi Zelnik-Manor. Approximate nearest subspace search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(2):266–278, 2011.

[18] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *STOC*, pages 537–546, 2008.

[19] Sariel Har-Peled and Nirman Kumar. Approximate nearest neighbor search for low dimensional queries. In *SODA*, pages 854–867, 2011.

[20] Sébastien Collette, Vida Dujmovic, John Iacono, Stefan Langerman, and Pat Morin. Distribution-sensitive point location in convex subdivisions. In *Proc. SODA*, pages 912–921, 2008.

[21] Sunil Arya, Theocharis Malamatos, and David M. Mount. Space-time tradeoffs for approximate nearest neighbor searching. *J. ACM*, 57(1), 2009.

[22] Sunil Arya, Theocharis Malamatos, David M. Mount, and Ka Chun Wong. Optimal expected-case planar point location. *SIAM J. Comput.*, 37(2):584–610, 2007.

[23] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[24] Vladlen Koltun and Micha Sharir. Polyhedral voronoi diagrams of polyhedra in three dimensions. In *Proc. Symp. Comp. Geom.*, pages 227–236, 2002.

[25] Yusu Wang. Approximating nearest neighbor among triangles in convex position. *Inf. Process. Lett.*, 108(6):379–385, 2008.

[26] M. Mitzenmacher and E. Upfal. *Probability And Computing: Randomized Algorithms And Probabilistic Analysis.* Cambridge U. Press, 2005.