Randomly-oriented RKD-trees

by

Dimitri N. Nicolopoulos

B.Sc., University of British Columbia

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

Graduate Program in Logic, Algorithms and Computation

at the

NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

March 2014

Author .....
Department of Mathematics
March 19, 2014

Randomly-oriented RKD-trees

by

Dimitri N. Nicolopoulos

Submitted to the Department of Mathematics
on March 19, 2014, in partial fulfillment of the
requirements for the degree of
Graduate Program in Logic, Algorithms and Computation

Abstract

Consider a set S of points in a real D-dimensional space $\mathbb{R}^D$, where distances are defined using function $\Delta : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ (the Euclidean metric). Nearest neighbor search is an optimization problem for finding the closest points in S to a given query point $q \in \mathbb{R}^D$. Given a positive real $\epsilon > 0$ then a point $p \in S$ is a $(1+\epsilon)$-*approximate nearest neighbor* of the query point $q \in \mathbb{R}^D$ if $dist(q,p) \leq (1+\epsilon)dist(q,p_{nn})$ where $p_{nn} \in S$ is the true nearest neighbor to q. If the data that is expressed in high-dimensional space $\mathbb{R}^D$ lies closer to an embedded manifold $\mathcal{M}$ of dimension d, where $d \ll D$, then, we show the data may be preprocessed into the Randomly-oriented RKD-trees structure and we provide a near optimal bound on the number of levels required to reduce the size of its cells by a factor $s \geq 2$. We show the data may be preprocessed into the structure in $\mathrm{O}\big(D \cdot N \cdot \log N\big)$ time and $\mathrm{O}\big(D \cdot N\big)$ space, so that given a query point $q \in \mathbb{R}^D$ and $\epsilon > 0$, a $(1+\epsilon)$-approximate nearest neighbor of q may be reported in $\mathrm{O}\big(\big(\frac{D}{\epsilon}\big)^{\mathcal{O}\left(d \log d \log\left(d\frac{D}{\epsilon}\right)\right)} \cdot \log N\big)$ time. Following the theoretical results, we show that the methods presented offer a highly efficient implementation in high-dimensional ANN search. Our implementation extends the *Computational Geometry Algorithms Library* (CGAL) and more specifically the spatial searching package of the library. The experimental results show that the proposed algorithm offers a state of the art ANN search algorithm when searching among high-dimensional data and high-dimensional data with an underlying low-intrinsic dimensional subspace.

Thesis Supervisor: Ioannis Z. Emiris
Title: Professor

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The *nearest neighbor search* (NNS) problem is: given a set S of N data points in a metric space X with distance function Δ, preprocess the data so that, given a query point $q \in X$, we can quickly determine the data point $p \in S$ that is closest to q. The problem was first introduced by Donald Knuth [22], referring to it as the *post-office problem*. It has attracted the development of numerous algorithms. The increase in the descriptive power offered by the analysis of high-dimensional datasets has placed a proportional burden on the computational expense exhibited by nearest neighbor search algorithms. Applications in areas such as machine learning [8], computational geometry [28] and data mining [15] place high importance on efficiently solving such spatial search problems. The prevalence of high-dimensional data and its descriptive power has enticed researches with the expectation of a large 'return'. Effective approaches have been constructed to deal with low-dimensional nearest neighbor search [14], yet the same cannot be said for its high-dimensional equivalent. The problem mainly stems from the inability to effectively deal with the "*curse of dimensionality*" [25]. For large values of dimension D we have not successfully produced any algorithms that solve the problem more efficiently than the brute-force method.

Solving the problem using the brute-force search method takes $\mathrm{O}\big(D \cdot N\big)$ time. Modest constant factor improvements have been proposed using methods shown in [9], and other orthogonal projection methods like that of Friedman et al. [17]. We place

our focus on methods using spatial data structures small enough to be stored in memory. There is a significant amount of literature dealing with the nearest neighbor search problem, see for example the work of Clarkson [7], Roussopoulos et al. [27].

For uniformly distributed point sets, using spatial partition trees to recursively subdivide space into increasingly fine convex sets. Friedman et al. [18] generalized older results showing that a structure may be built using $\mathrm{O}(N)$ space and query search may be performed in $\mathrm{O}(\log N)$ time in the expected case. The methods proposed the use of kd-trees. However even this method does not escape the negative effect dimensionality increase plays in the hidden constant factors. These are shown to increase as much as 2^D.

Highly effective structure representations, designed to solve the NNS optimization problem in *high-dimensions*, have been exploited in all sorts of manner: BBD-trees [4], Voronoi diagrams [2], RP-trees [10], PCA trees [30] and locality-sensitive hashing functions [11]. Such structures have been established to be used within a NNS algorithm given one common assumption, we place aside the *exact* nearest neighbor search problem and focus on obtaining an approximate result (see Definition 1). The optimization of search times allows for the acceptance of a small error ($(1+\epsilon)$-approximate) in comparison to the exact result (see Figure 1-1). That is to say, we accept an approximation of the true nearest neighbor by tolerating an imperfect result, for a gain in speed and memory savings, particularly in situations where the set of points lies in a high-dimensional space.

Definition 1 (Approximate nearest neighbor search)**.** *Given a set $P = \{p_1, \ldots p_N\} \in X$ of dimension D in a metric space $M = (X, \Delta)$, process the points into a data structure of size $poly(N, D)$, in time $poly(N, D)$ such that: For any query point $q \in X$ and radius $r \in \mathbb{R}$, if there exists $p \in P$ such that $\Delta(p, q) \leq r$, find a point $p' \in P$ such that $\Delta(p', q) \leq c \cdot r$ for a constant c.*

Increasing ubiquity of high-dimensional data sets has focused research on effectively tackling obstacles of data sparsity and dissimilarity faced in high dimensions. Like many data structures, k-d trees [6], [17] display weaknesses successfully address-

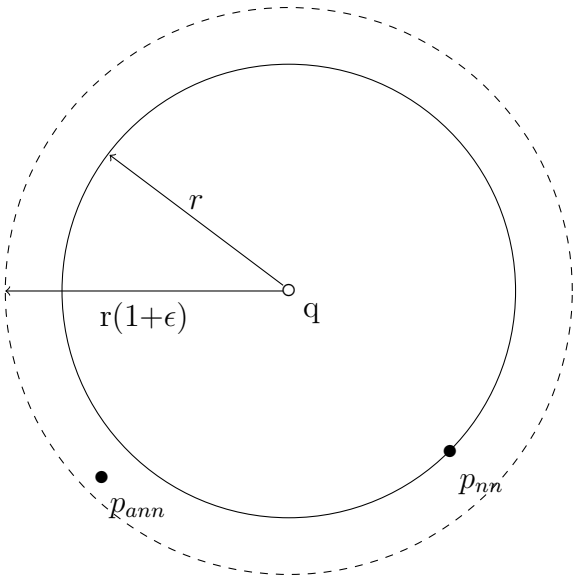


Figure 1-1: Approximate nearest neighbor

ing the curse of dimensionality [20]. The problem of approximately nearest neighbor has also been considered by Arya and Mount [3], where they proposed a randomized data structure that achieves polylogarithmic query time in the expected case, and close to linear space. They later propose, in [4] an algorithm with query time $\mathrm{O}\big(D\lceil 1+6D/\epsilon\rceil^{D}\cdot\log N\big)$ and space $\mathrm{O}\big(D\cdot N\big)$. In their algorithms, as in ours the approximation error is an arbitrary positive constant, fixed in prepossessing time. Both the values of D and ϵ are independent of the data size N. We attempt to strengthen the result when we are in possession of additional information regarding data set.

It has recently become evident that speed and space gains may be addressed by exploring an additional avenue. Data ostensibly lying in high-dimensional space $\mathbb{R}^D$, may have a low intrinsic dimension, lying closer to a manifold of dimension $d \ll D$ (see [10]). We are interested in algorithms leveraging both the approximation of results and an automatic adaptability to the intrinsic dimension of the dataset in the pursuit of efficiently solving the $(1+\epsilon)$-approximate nearest neighbor problem.

Theorem 2. *Consider a finite set $S \subseteq \mathbb{R}^D$ of data points, of cardinality $|S| = N$ and Assouad dimension d. There is a constant $c_{d,D,\epsilon}$ that depends on the dimension D the intrinsic dimension d and the error ϵ where $c_{d,D,\epsilon} = \mathcal{O}\left(\left(\frac{D}{\epsilon}\right)^{\mathcal{O}\left(d\log d\log\left(d\frac{D}{\epsilon}\right)\right)}\right)$,*

such that in $\mathcal{O}(DN \log N)$ *time it is possible to construct a data structure of size* $\mathcal{O}(D \cdot N)$*, such that for the* L_1 *metric:*

- *Given any* $\epsilon > 0$ *and query point* $q \in \mathbb{R}^D$*, a* $(1+\epsilon)$*-approximate nearest neighbor of* $q \in S$ *can be reported in* $\mathcal{O}(c_{d,D,\epsilon} \cdot \log N)$

The Theorem shows that the space requirements are independent of ϵ. Once the data structure is built ϵ may change without rebuilding the structure. Also, we find that the dependency to D is logarithmic in the exponent of the constant factor of the query time, in comparison BBD-trees has a linear dependency on the same exponent. Finally, a relation to the intrinsic dimension d of the data set is evident on the query time factor.

1.1 Previous Work

1.1.1 Assouad Dimension (Doubling Dimension)

One of the definitions of the intrinsic dimension, serving our purposes, as it appears in [10], is the Assouad (or doubling) dimension [5].

Definition 3 (Assouad dimension)**.** *For any point* $x \in \mathbb{R}^D$ *and any* $r > 0$*, let* $B(x,r) = \{z : \|x - r\| \leq r\}$ *denote the closed ball of radius* r *centered at* x*. The Assouad dimension of* $S \subset \mathbb{R}^D$ *is the smallest integer* d *such that for any ball* $B(x,r) \subset \mathbb{R}^D$*, the set* $B(x,r) \cap S$ *can be covered by* 2^d *balls of radius* $r/2$*.*

Systematic investigation in the area largely attempts to exploit low intrinsic dimensional structure found in common data of interest.

1.1.2 RP-tree

RP-trees are a k-d tree variant presented by Dasgupta and Freud [10]. They possess the property of intrinsic dimension adaptation. Dasgupta and Freud manage to show the validity of the following statement: given a cell C in the RP-tree. If the data

in C have intrinsic dimension d then all the descendant cells $d \log d$ or more levels below will have at most half the diameter of C. The conclusion derived is that there is no dependence on the extrinsic dimensionality (D) of the data. The important distinction from k-d trees comes from the fact that the data is not separated by coordinate parallel hyperplanes but by a direction orthogonal to a segmentation that is picked uniformly at random from the unit sphere S^{D-1} at each level of the resulting tree.

1.1.3 RKD-trees

Silpa-Anan & Hartley [29] proposed a spatial data structure similar to k-d trees. *RKD-trees* evolved from randomized k-d trees of [23]. They attempt to take advantage of underlying structure of commonly used datasets in computer vision. Additional randomization is introduced on the choice of hyperplane, that splits space on each node of the binary trees. Backtracking optimization is exhibited with the use of multiple trees being searched with optimum heap sort structures.

M. Muja & D. Lowe [26] presented a highly efficient implementation aimed at solving queries in high-dimensional datasets. The results demonstrate one of the most competitive approximate nearest neighbor search options, particularly for higher dimensional space. The gain of searching for approximate solution does not necessarily mean a large loss in precision, in comparison to linear-search for exact NN points in S.

The RKD-tree is a powerful tool on a large variety of high-dimensional investigations. However, Dasgupta & Freund [10] exhibit that k-d tree like structures (splitting along coordinate dimensions) do not adapt to the intrinsic dimension of the data. In general situations this might not be a big deterrent, yet, given the choice, from the perspective of geometry, we tend to be more inclined to placing a high value on some theoretical bounds.

1.1.4 K-d trees do Not Adapt to Intrinsic Dimension

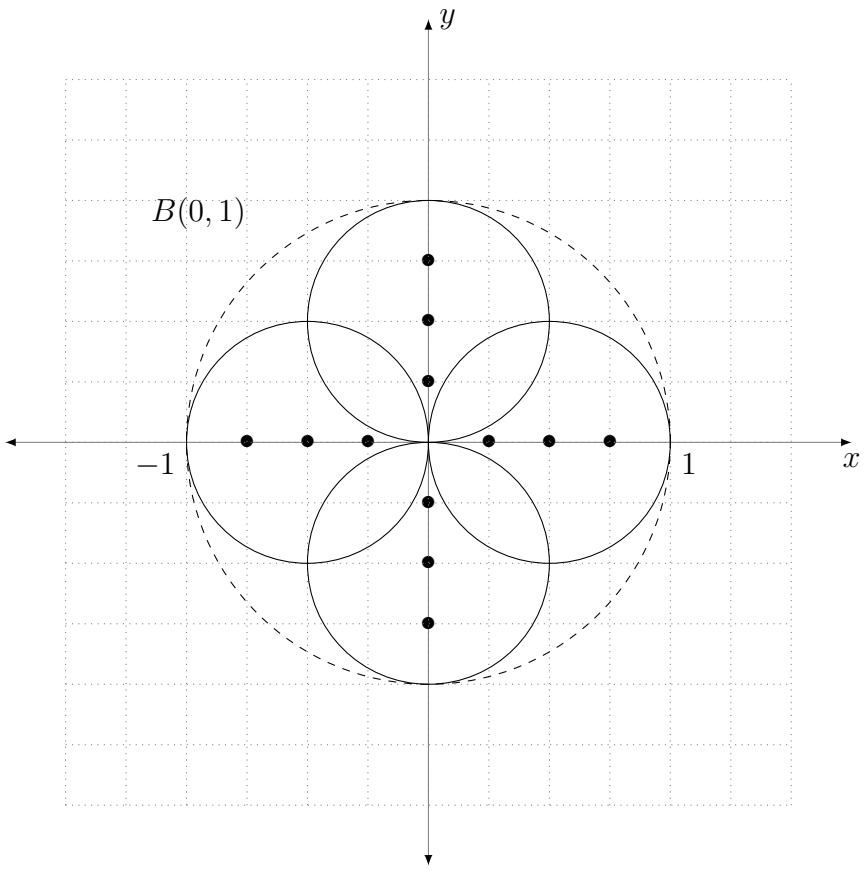


Figure 1-2: The set of points defined in $S = \bigcup_{i=1}^{2} \{\mathrm{te_i} : -1 \leq t \leq 1$ can be covered by 2D balls of half the radius. The Assouad dimenson of S is $2^d = 2D \Rightarrow d = \log(2D)$.

The following counterexample, depicted in [10], illustrates that k-d trees do not adapt to intrinsic dimension: if we consider $S \subset \mathbb{R}^D$ made up of the coordinate axes between -1 and 1 then $S = \bigcup_{i=1}^{D} te_i : -1 \leq t \leq 1$. Where e_i for $i = 1, \ldots D$ is the canonical basis of $\mathbb{R}^D$. S lies within ball $B(0, 1)$ where the center $x = 0$ and radius $r = 1$, can be covered by $2D$ balls of half the ball radius $r/2 = 1/2$ (see figure 1-2). The Assouad dimension of S is the smallest integer d such that for any ball $B(x, r) \subset \mathbb{R}^D$ the set can be covered by 2^d balls of radius, thus it is $2^d = 2D \Rightarrow \log 2^d = \log 2D \Rightarrow d = \log 2D$.

1.1.5 Randomly-oriented k-d Trees Adapt to Intrinsic Dimension

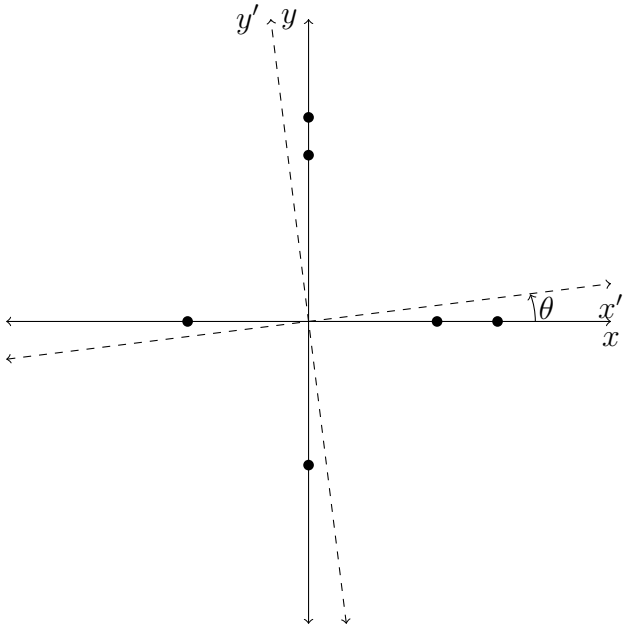


Figure 1-3: Rotating the basis by some random θ, 2-dimensional basis

An advantage of k-d trees not retained by RP-trees in approximate nearest neighbor search (ANNS), is that the search time increases at each level of the tree by at least $\mathrm{O}(n)$ (produced by a dot product providing the resulting projection at each level). Vempala successfully preserves the eloquent simplicity of axis-parallel cuts of k-d trees. In his paper 'Randomly-oriented k-d Trees Adapt to Intrinsic Dimension' [32], Vempala creates a new random space, a random rotation of the ambient space (see figure 1-3), prior to building the k-d tree on the new space. The method is shown to adapt to the intrinsic dimension and at the same time retain the advantages offered by k-d tree in ANNS search.

RP-trees provide a robust and simple to implement algorithm. They adapt to the intrinsic dimension of the data and, without a high degree of difficulty, may exploited effectively in approximate nearest neighbor search. Despite the promise of results, there seems to be limited literature supporting the use of RP-trees as a state of the art option in ANNS. RP-trees do not seem to pose a viable alternative compared to

efficient ANNS algorithms that are extensively proven in practice (see [26], [11], [26] etc.)

Vempala introduces Randomly-oriented k-d trees; making claim of the first orthogonal hyperplane splitting structure that adapts to the data's intrinsic dimension. Randomly-oriented k-d trees have not yet been implemented and do not seem to lend themselves to a state of the art algorithm. Although, the process of randomly rotating the ambient space may be extracted and used in other algorithms. We provide such an algorithm, combining the aforementioned extracted process with the high performance RKD-trees implementation of M. Muja & D. Lowe [26]. The algorithm is (1) shown to compare, in practice, to other state of the art implementations in regards to search time in ANNS optimization problems and (2) provably adapt to the intrinsic dimension of the dataset.

1.1.6 Computational Geometry Algorithms Library

The Computational Geometry Algorithms Library (CGAL) [1] is a software library aiming to provide access to efficient and reliable algorithms in computational geometry. The majority of algorithms focus on 2 and 3 dimensional datasets, but a lately a larger emphasis is placed in higher D dimensional data. The structures and algorithms operate on geometric objects corresponding to points and space segments. The current nearest neighbor algorithm search is located within the dD Spatial Search package.

1.2 Randomly-oriented RKD-trees

We introduce *Randomly-oriented RKD-trees*, and show that the data structure adapts to intrinsic dimension. As a high-dimensional ANNS algorithm it is shown, in practice, as a high-performance option. The implementation utilizes the computational geometry algorithms library (CGAL). Our algorithm is variation of the randomized k-d trees implemented and presented by M. Muja and D. Lowe [26], essentially preprocessing the data, so that the search time is unaffected. The build time contains an

added matrix multiplication dependent on the size N and dimension D. We generate a random rotation matrix, randomly rotating the ambient space, similarly to [32]. A query point is mapped to the new basis adding $O(D \log N)$ to $O(D \log N)$ run time. The main theorem we present provides a strong guarantee for Randomly-oriented RKD-trees, that they adapt to intrinsic dimension, similarly to Vempala's assertion and RP-trees.

Theorem 4 (adapted from Theorem 3 [10] and Theorem 2 [32])**.** *Let $S \subset \mathbb{R}^D$ be a finite set with n points and Assouad dimension d. Let $D \log D \leq c_0 N$ for the Randomly-oriented RKD-Tree, with probability at least $1 - n\mathrm{e}^{-c_1 N}$ for any cell C of the tree and every cell C' that is at least $c_2 D \log D$ levels below C, we have $diam(C' \cap S) \leq \frac{1}{2} diam(C \cap S)$ where c_0, c_1 and c_2 are absolute constants.*

Continuing in the same fashion as shown in Vempala's Randomly-oriented k-d trees, and focusing more on creating a state of the art high-dimensional ANNS implementation, the theorem shows that simply randomly rotating the data may yield advances in existing k-d tree like algorithms meant for ANN search in high-dimensions. Axis-parallel cut structures are further supported as a structure of choice when the intrinsic dimension is lower than the ambient dimension in search optimization problems.

1.2.1 Contributions

Typical guarantees given by data structures like k-d trees [6] and BBD trees [4] are:

Space Partitioning Guarantee: There exists a bound $L(s)$, $s \geq 2$ on the number of levels one has to go down before all descendants of a node of size Δ are of size Δ/s or less. The size of a cell is variously defined as the length of the longest side of the cell.

Bounded Aspect Ratio: There exists a certain "roundness" to the cells of the tree - this notion is variously defined as the ration of the length of the longest to the shortest side of the cell, ratio of the radius of the smallest circumscribing ball of the cell to that of the largest ball that can be inscribed in the cell.

Packing Guarantee: Given a fixed ball B of radius R and a size parameter r, there exists a bound on the number of disjoint cell of the tree that are of size greater than r and intersect B. Such bounds are usually arrived at by first proving a bound on the aspect ratio for cells of the tree.

Noting that the listed guarantees are also the main focus of [12], we place the same significance in showing that the high-performance approximate nearest neighbor algorithm we present is adherent to the guarantees placed and proved later in the paper.

We start by defining the Randomly-oriented RKD-trees ANNS algorithm, we then present the arguments demonstrating the above guarantees. More specifically, we present a bound on the number of levels required for reducing the size in a Randomly-oriented RKD-tree – subsequently the same bound is also shown for Randomly-oriented k-d tree – not just by 2, but by any given factor s. Our result improves upon the results presented in [32] and shows how a minor modification in the RKD-tree [26] pre-processing step can guarantee low-dimensional manifold adaptation. Next we provide an aspect ratio for both structures. These results are not provided in [32] and [26]. Due to the randomized nature of the algorithms these are difficult to bound in a clear-cut fashion but are useful results in providing a packing lemma for both.

The paper is heavily reliant on the structure and advances made in the papers of Dasgupta and Freud [10], Vempala [32], Dhesi and Kar [12] and on the implementation provided by M.Muja and D.Lowe [26]. We make similar assertions that fit our needs and obtain good probability of the adaptation of the algorithm to the low manifold when that manifold is considerably lower than the ambient one.

We demonstrate that the proposed data structure retains the experimental efficiency in search time depicted by RKD-trees. We exhibit that the adaptation to low intrinsic dimension combines with an aspect ratio bound to produce a packing lemma directly. The result associates the intrinsic dimension with a bound on the ANNS algorithm's search time. Finally, we aim to propose its usability on an open source and long-established application framework, namely by adding the Randomly-oriented RKD-trees structure and the associated ANNS algorithm to the Computational Ge-

ometry Algorithms Library (CGAL).

Chapter 2

Randomly-oriented RKD-trees

In this chapter, we introduce *randomly oriented randomized k-d trees* or *Randomly-oriented RKD-trees*, which is the data structure used in our approximate nearest neighbor algorithm. It is an instance of geometric data structures defined by the property of hierarchical decomposition of space by axis-aligned hyperplanes. The main distinctive attribute of Randomly-oriented RKD-trees is that the structure is shown to adapt to the intrinsic dimension d of the data set, something that is shown to play a role in the determination of the $(1+\epsilon)$-approximate nearest neighbor query time.

The data structure recursively subdivides space by axis-aligned hyperplanes as evenly and time efficiently as possible, given the randomization that is introduced. The expectation is, similarly to the optimized kd-tree [17], [6], that the *cardinality* of points associated with the nodes on any path in the tree decreases exponentially. In contrast to the kd-tree, RKD-trees [29], [26] introduce some randomization to each tree thus removing the optimality of balance at each level. The creation of multiple trees then exploits the variation in each tree's artificial imbalance. The Randomly-oriented k-d trees (see Vempala [32]) randomly rotate the space S the points lie in. The important feature of Vemplala's structure is that it contains the properties of axis-alignment in hyperplane direction, yet at the same time is shown to adapt to the doubling dimension (intrinsic dimension) of the data set. The Randomly-oriented RKD-trees structure is formed on a combination of the previous ideas. It is made

to achieve high-performing ANN search results in high dimensional space, taking advantage of fast backtracking methods. It is also formed to take advantage of the intrinsic dimension of data set with respect to ANN query search.

Randomly-oriented RKD-trees are analogous to balanced tree structures based on orthogonal partitioning of space. The K-d tree structure [6] generalizes the binary tree to high-dimensional space. Optimized k-d tree [18] introduces a logarithmic search-time, the logarithmic search-time does not however extend to high-dimensional search. K-d trees are an effective structure when used in nearest neighbor algorithms in low dimensions, however their efficiency diminishes for high-dimensional data. The RKD-tree structure [29] in introduces randomization on parameters, selecting the partitioning value in a random manner, while structures such as k-d trees [6] select the partitioning dimension in a cyclical order. Randomly-oriented RKD-trees are also similar to other structures such as Randomly-oriented k-d tree [32], the rotated tree structure presented in [29], which rotate the space to create different structures through which to search. We will show that it is possible to create a structure which combines features of all the previous structures so that it adapts to the intrinsic dimension of the data set and at the same time performs highly efficient ANN search in high-dimensional space by preserving backtracking efficiency.

2.1 Overview of the Construction Process

What can be thought of as the precursor to the construction of Randomly-oriented RKD-trees is a random rotation of the space. The structure is then constructed through the repeated application of splits, either median or mean. However, we have not tested both within the same structure. The types of splits represent the way of subdividing a cell into two smaller ones called its *children*. A split partitions a cell by an axis-orthogonal hyperplane. The splitting coordinate along which the data is split is called the *cut-dimension*. The value separating the values along the cut-dimension is called the *cut-value*. The two children are called the *left child* and *right child* depending on whether the coordinates along the cut-dimension are less than or

greater than the cut-value of the splitting plane.

The Randomly-oriented RKD-trees is constructed through a series of splits. Given a set S of N data points in $\mathbb{R}^D$. The root of each of the Randomly-oriented RKD-trees is a node whose associated cell is associated set is the entire set S. The recursive construction algorithm is given a cell and a subset of data points associated with this cell. Each stage of the algorithm that creates the structure determines how to subdivide the current cell though splitting and then partitions the points among the child nodes. The process is repeated until each cell on the level is associated with at most one point. These nodes are called the *leafs*.

2.2 Randomly Rotating the Space

Prior to building any structure on the given data set, an essential process relating to the structures adaptability to the intrinsic dimension must first be performed. The process dictates we select a random orthogonal basis, then build an RKD-tree using this basis. That is, we create a rotation matrix in D-dimension and multiply it by a matrix representing the data set S. Using the subgroup algorithm of Diaconis & Shashahani [13], we recursively generate a rotation matrix. The initial step involves the construction of a 2×2 rotation matrix. Then, each step i to $i + 1$, involves the generation of a vector v uniformly distributed on the i-sphere, in $\mathrm{O}\big(i\big)$ time at each step. A rotation at each step is embedded into a matrix of the next larger size such that it generates the rotation of the current size. Each such step employs the use of the Gram-Schmidt process [33], and has a time complexity of $\mathrm{O}\big(D \cdot i^2\big)$ at each step. A matrix multiplication is performed at each step, which we naively say contributes $\mathrm{O}\big(i^3\big)$ (more sophisticated methods are employed for matrix multiplication in the implementation). The process continues until a rotation matrix is created, of size equal to the dimension D of the data set.

2.3 Partitioning the Points

Prior to describing how the splitting algorithms operate, we describe how the points are partitioned at each level of the tree. We apply the same method as the RKD-trees implementation found within the *Fast Library for Approximate Nearest Neighbors* (FLANN) library of Muja & Lowe [26]. We assume that the data points that are associated with the current node are stored in a vector of size N. Each point is associated with a D dimensional array. The initial tree containing all the data points is built in $\mathrm{O}\big(K \cdot N \cdot \log N\big)$ time, a cut-value is determined in time $\mathrm{O}\big(K\big)$ which depends on the splitting algorithm used. The points on each level are separated into the children in $\mathrm{O}\big(N\big)$ time, on a tree of size $\mathrm{O}\big(\log N\big)$.

To partition the points, we determine the cut-value for each cut-dimension at which the points will be separated. At each node and in $\mathrm{O}\big(N\big)$ time the points are placed into two separate containers, a left child and a right child depending on whether the value of the cut-dimension coordinate of a point is less than or greater than the cut-value.

We complete the overview of the construction algorithm by describing the two splitting algorithms we employ for creating the hyperplane of each partition. The two methods are the *mean split algorithm* [26] and the *median split algorithm* [32]. Both approximate the true values mean or median with a different approach. This bounded random deviation from the true values provides assurance that with high probability no two trees created are the same. The variance also plays an integral role in the proof that the structure adapts to the intrinsic dimension of the data set.

2.4 Mean Split

The mean algorithm is characterized by the speed at which a fairly balanced tree can be derived. A cell within a tree is any cell that has been created by a recursive application of the *mean split rule*, starting from an initial cell, i.e. the root of the tree, that is associated with all the points in the data set.

Mean split rule. The main incentive here is to produce a balanced split at each node by calculating both the cut-dimension and cut-value in a time efficient manner. At the same time a positive side effect is: given any cut-dimension the same points will not be separated into two child nodes in exactly the same fashion. Let $m \in \mathbb{N}_{>0}$, where $m \leq N$, the sample size for which the mean and variance is determined for each dimension in the data set. We calculate the variance off a uniform random sample of each dimension, in time $\mathrm{O}\big(mD\big)$ which for small m acts more like $\mathrm{O}\big(D\big)$. The dimension indexes are sorted in accordance to the value of the variance. From a subset of size $m_v \in \mathbb{N}_{>0}$, where $m_v \ll D$ an index associated with the dimensions of the highest variance is picked uniformly at random. The index returned from the process is the cut-dimension. Then the cut-value is determined as the mean associated with the cut-dimension.

2.5 Median Split

The median algorithm's main function is to provide the theoretical guarantees needed for the main proof. It is also implemented, yet not preferred to the mean algorithm when construction time is of higher importance. By using the median algorithm in our proof we avoid the added complication of taking each dimensions' distribution into account.

Median split rule. As with the previous split algorithm, balancing the tree is of importance. Let $m \in \mathbb{N}_{>0}$, where $m \leq N$, the sample size for which variance is determined for each dimension in the data set. We calculate the variance off a uniform random sample of each dimension, in time $\mathrm{O}\big(mD\big)$ which for small m acts more like $\mathrm{O}\big(D\big)$. The dimension indexes are sorted in accordance to the value of the variance. From a subset of size $m_v \in \mathbb{N}_{>0}$, where $m_v \ll D$ an index associated with the dimensions of the highest variance is picked uniformly at random. The index returned from the process is the cut-dimension. The implementation then calls for calculating the diameter Δ of the bounded set $S' \subseteq S$ that is associated with any given cell in the tree. The difference here is that the cut-value is determined by

determining the value of the median along the cut-dimension and then selecting a value δ uniformly at random that lies within a range $[-\frac{6\Delta}{\sqrt{D}}, \frac{6\Delta}{\sqrt{D}}]$. The range chosen is more a technical matter which is clarified in the next section.

2.6 Multiple Trees

Finally, a main distinction from other more classically built data structures is that RKD-trees, and subsequently Randomly-oriented RKD-trees, manage to offer a structure that may utilize a k number of trees when used within a process. As described earlier, each tree has been built by *injecting* some randomization as to produce a different tree each time. The usefulness, in relation to ANN search, relates to achieving highly efficient backtracking performance. When each tree is searched initially one by one down to the leaf, a sorted heap containing the best subtree not visited is built containing nodes from all trees. The best ones are then traversed up until the $1 + \epsilon$ approximation rule is achieved. The effectiveness of the method is currently exemplified within the experimental results, and does not play a role in the theoretical guarantees provided in the next section.

Chapter 3

Size Reduction Theorem

3.1 Extending Theorem 4

The Randomly-oriented k-d Tree structure adapts to doubling dimension of data (see [32], section 3.4). According to Theorem 22 of [10] low-dimensional manifolds have low doubling dimension (see definition 3), thus the structure adapts to manifold dimension. The following conclusions, more importantly, show an adaptation of the Randomly-oriented RKD-tree structure to the doubling dimension of a given dataset.

We extend the result of Theorem 2 of [32], bounding the number of levels needed to decrease the size of a given cell by a factor of $s > 2$. Given we have data of a set S in a cell C of radius Δ, after $c_1 d \log d$ levels of partitioning the resulting cell will have a radius lesser than or equal to $\frac{\Delta}{2}$. It is argued in [12] that given the large number of nodes at the $c_1 d \log d$ levels and the fact that the success probability in 4 is just above a constant bounded away from 1, it is not possible to argue that after another $c_1 d \log d$ levels of partitioning the descendants will have a radius $\leq \Delta/4$. Extending the result found in [32] and adapting it to the Randomly-oriented RKD-tree (and Randomly-oriented k-d Tree) structure we have:

Theorem 5 (Extension of Theorem 2 in [32], taken from [12])**.** *For any $\delta > 0$, with probability at least $1 - \delta$, every descendant $c^{'}$ which is more than $c_1 d \log d + \log(1/\delta)$ levels below C has $radius(C^{'}) \leq radius(C)/2$.*

We use arguments presented in [12], showing that the same enhancements will work for Randomly-oriented RKD-tree (and Randomly-oriented k-d Tree) structures. Taking cell C of radius Δ go down $L = c_1 d \log d + 2$ levels to 2^L nodes. Then a further $L^{'} = c_1 d \log d + L + 2$ levels below. The probability of any of these descendants is greater than $\frac{\Delta}{4}$ after $L^{'}$ levels is less than $\frac{1}{4 \cdot 2^{L'}}$. So with probability at least $1 - \frac{1}{4} - \frac{1}{4 \cdot 2^{L'}} \cdot 2^{L'} \geq \frac{1}{2}$ all descendants after $L^{'}$ levels of partitioning will have radius $\leq \frac{\Delta}{4}$. This argument is formulated in the following Theorem:

Theorem 6 (taken from [12])**.** *There is a constant c_2 with the following property. For any $s \geq 2$, with probability at least $1 - \frac{1}{4}$, every descendant $C^{'}$ which is more than $c_2 \cdot s \cdot d \cdot \log d$ levels below C has $radius(C^{'}) \leq radius(C)/s$.*

Proof. Assume s is a power of 2. Proving by induction and taking as base case ($s = 2$) the argument presented in Theorem 5. Induction step: let $L(s)$ be the number of levels it takes to reduce the size by a factor of s with high confidence. Then:

$$L(s) \leq L(s/2) + c_1 d \log d + L(s/2) + 2 = 2L(s/2) + c_1 d \log d + 2.$$

Continue by solving the recurrence,

$$\begin{aligned}
L(s) &\leq 2L(s/2) + c_3 d \log d + 2 \\
&\leq 2(2L(s/2^2) + c_3 d \log d + 2) + c_3 d \log d + 2 \\
&\qquad = 2^2 L(s/2^2) + 2c_3 d \log d + 2^2 + 2 \\
&\leq 2^2(2L(s/2^3) + c_3 d \log d + 2) + 2c_3 d \log d + c_3 d \log d + 2^2 + 2 \\
&\qquad = 2^3 L(s/2^3) + 2^2 c_3 d \log d + 2c_3 d \log d + c_3 d \log d + 2^3 + 2^2 + 2 \\
&\ldots \\
&\leq 2^i L(s/2^i) + d \log d \cdot \left(\sum_{j=0}^{i-1} 2^j \right) + \left(\sum_{j=1}^{i} 2^j \right)
\end{aligned}$$

For $i = \log s \Rightarrow L(s) = 2^{\log s} L(s/2^{\log s}) + c_3 d \log d \cdot \left(\frac{1 - 2^{\log s}}{1-2} \right) + \left(\frac{\log s \cdot (1 - 2^{\log s})}{1-2} \right)$. Thus, $L(s) = \mathrm{O}\big(sd \log d\big)$. □

3.2 Generalizing the size reduction theorem

The extensions of Theorem 4 lay the groundwork for the generalization we prove in this section.

Theorem 7 (Adapted from Theorem 5 in [12]). *There is a constant c_3 with the following property. Suppose a Randomly-oriented RKD-Tree (or Randomly-oriented k-d Tree) is built using data set $S \subset \mathbb{R}^D$. Pick any cell C in the Randomly-oriented RKD-Tree; suppose that $S \cap C$ has doubling dimension $\leq d$. Then for any $s \geq 2$, with probability at least $1 - \frac{1}{4}$ (over the choice of randomization in constructing the subtree rooted at C), for every descendant C' which is more than $c_3 \cdot \log s \cdot d \log sd$ levels below C, we have $radius(C') \leq radius(C)/s$.*

The result guarantees a reduction for a logarithmic factor. We outline the framework of the proof, and set the foundation of the main theorem which we prove later on. Suppose the data set $S \subset \mathbb{R}^D$ has doubling dimension d, which is used to build the trees. Let C be some cell of the tree. If $S \cap C$ lies in a ball of radius Δ, what we aim to show is that after $\mathrm{O}\big(\log s \cdot d \log sd\big)$ levels of splitting, starting from cell C, each of the resulting cells is contained in a ball of radius $\leq \Delta/s$.

Cover $S \cap C$ with N balls $B_1,\ B_2,\ , \ldots, B_N$ of radius $\mathrm{O}\big(\frac{\Delta}{s \cdot \sqrt{d}}\big)$. Fix two balls B_i and B_j at a distance greater than $(\Delta/s) - (\Delta/(s \cdot \sqrt{d}))$. Then, a projection orthogonal to all the basis vectors used for cuts so far has a constant probability of separating the balls, completely, from one another. N^2 pairs of balls exist, so after $\Theta(d \log d)$ levels below C, in each tree, each cell contains points from B_i that is within a distance $(\Delta/s) - (\Delta/(s \cdot \sqrt{d}))$ of each other. Thus the radius of the cells is $\leq \Delta/s$.

Randomly-oriented RKD-trees's method of choosing a random basis relies on picking a series of orthogonal unit vectors to each subsequent formed basis. Its success in this case is dependent on the relation of the dataset size to the dimension size. There must exist an exponential dependence between the two, given the dataset size is greater than the dimension size. RP-trees rely on the independent randomness of each projection, we are able to achieve a 'pseudo' independent randomness. But 'pseudo', as is shown in the following sections, is enough.

3.3 Precursor

The proof relies on the use of a very effective tool in such circumstances, the Johnson-Lindenstrauss Lemma [21]. We use the same version presented in [10]. Let V be a subspace of $\mathbb{R}^D$, and $\pi_V(.)$ an orthogonal projection to V.

Lemma 8. *Fix a unit vector $u \in \mathbb{R}^D$, let V be a random k dimensional subspace where $k < D$, and $\epsilon > 0$ then:*

$$\mathbb{P}(\|\pi_V(u)\| > (1+\epsilon)\tfrac{k}{D}) \leq e^{-\frac{k}{4}(\epsilon^2-\epsilon^3)}$$

$$\mathbb{P}(\|\pi_V(u)\| < (1-\epsilon)\tfrac{k}{D}) \leq e^{-\frac{k}{4}(\epsilon^2-\epsilon^3)}$$

Proof. See Appendix. □

Thus for any finite set of points $S \subset \mathbb{R}^D$, with probability at least $1-2\binom{|S|}{2}e^{-\frac{k}{4}(\epsilon^2-\epsilon^3)}$ the following inequality holds true

$\forall u, v \in S$,

$$(1-\epsilon)\tfrac{k}{D}\|u-v\|^2 < \|\pi_V(u-v)\|^2 < (1+\epsilon)\tfrac{k}{D}\|u-v\|^2$$

For $k = 1$:

Lemma 9. *Let $u \in \mathbb{R}^D$ and $v \in \mathbb{R}^D$ be a random unit vector. For any $\beta > 0$,*

(a) $\mathbb{P}(\|\pi_v(u)\| \leq \frac{\beta}{\sqrt{D}}\|u\|) \leq \alpha\sqrt{\frac{2}{\pi}}$.

(b) $\mathbb{P}(\|\pi_v(u)\| > \frac{\beta}{\sqrt{D}}\|u\|) \leq \frac{2}{\beta}e^{-\frac{\beta^2}{2}}$

Proof. See Appendix. □

Lemma 10 (Lemma 5 in [32])**.** *Let $S \subset B(x, \Delta)$ be a set of doubling dimension d. Let V be an arbitrary subspace of $\mathbb{R}^D$, v be a random unit vector orthogonal to V, $0 < \delta < 1$ and $\beta = \sqrt{2(d + \log(2/\delta))}$.*
Then for any $0 < \delta < 1$, with probability $> 1 - \delta$ over the choice of projection and $r = 3 \cdot \frac{\Delta}{\sqrt{D-\tilde{d}}} \cdot \sqrt{2(d + \log\frac{2}{\delta})}$, we have $\pi_v(S) \subseteq [\pi_v(x) - r, \pi_v(x) + r]$.

Proof. See Appendix. □

Most projected points of S onto $\mathbb{R}^1$ lie in a central interval of of radius at most $\mathrm{O}\big(\Delta/\sqrt{D}\big)$.

Lemma 11. *Suppose $S \subset \mathbb{R}^D$ is within a ball $B(x, \Delta)$. Choose $0 < \delta \leq 1$, $0 < \epsilon \leq 1$ such that $\delta\epsilon \leq \frac{1}{e^2}$. For a measure μ on the set S, with probability $> 1 - \delta$ over the choice of random projection onto $\mathbb{R}^1$, the points will all lie within $\frac{\Delta}{\sqrt{D}}\sqrt{2\log(\frac{1}{\delta\epsilon})}$ of projected point $\pi_{v(x)}$. A small fraction ϵ of $\pi_{v(S)}$ lies outside of that radius.*

Proof. See Appendix. □

Lemma 12. *Given a normal distribution, $\mu = median$.*

Proof. See Appendix. □

The median of the projected points also lies in the same interval. But the projected points lie on a univariately normal distribution of some projected vector $\pi_{v(S)}$, and from lemma 12 $\mu_{\pi_{v(S)}}$ equals median$(\pi_{v(S)})$. Setting $\epsilon = 1/2$.

Corollary 13. *For $S \subset B(x, \Delta)$, $\delta \in (0, 2/\epsilon^2]$ and a random unit vector $v \in \mathbb{R}^n$, with probability at least $1 - \delta$,*

$$\|\text{median}(\pi_{v(S)}) - \pi_v(x)\| \leq \frac{\Delta}{D}\sqrt{2\log(\tfrac{2}{\delta})}.$$

Lemma 14. *Let $S \subseteq B(x, \Delta)$, and $z \in B(x, \Delta)$. Let V be a $\bar{d}$-dimensional subspace of $\mathbb{R}^D$ with $\bar{d} < D/9$ and v be a random unit vector orthogonal to V. Then, with probability at least 0.95,*

$$\|\text{median}(\pi_v(S)) - \pi_v(x)\| \leq \frac{6\Delta}{\sqrt{D - \bar{d}}}$$

Proof. See Appendix. □

Lemma 15. *Let $B = B(z, \bar{r})$ be a ball contained inside a cell of radius Δ enclosing a set of doubling dimension d. A random split separates the data of the ball with probability at most $\frac{3\bar{r}\sqrt{d}}{2\Delta}$.*

Proof. A random projection line chosen from the basis $D - \bar{d}$ is the line to which the data is projected to, a line of which a split point is chosen within the interval of length $\frac{12\Delta}{\sqrt{D}}$. Note that the random direction is independently chosen to the split point. The probability that the data on the interval $\pi_v(B)$ of radius r is split is $\bar{r} \cdot \frac{\sqrt{D}}{6\Delta}$. Letting ρ be the random variable giving the radius of the interval $\pi_v(B)$ we have the probability of B getting split:

$$\begin{aligned}\frac{\sqrt{D}}{6\Delta}\int_0^\infty r\mathbb{P}[\rho = r]\mathrm{d}r &= \frac{\sqrt{D}}{6\Delta}\int_0^\infty\int_0^r \mathbb{P}[\rho = r]\mathrm{d}t\mathrm{d}r \\ &= \frac{\sqrt{D}}{6\Delta}\int_0^\infty\int_t^\infty \mathbb{P}[\rho = r]\mathrm{d}r\mathrm{d}t = \frac{\sqrt{D}}{6\Delta}\int_0^\infty \mathbb{P}[\rho \geq t]\mathrm{d}t\end{aligned}$$

The probability upper bound of $\mathbb{P}\left[\rho \geq \frac{3\bar{r}}{\sqrt{D-\bar{d}}}\sqrt{2\left(d + \ln 2\right)}\right] \leq \eta$ plus the upper bound of any probability $\mathbb{P}\left[\rho \geq t\right] \leq 1$ for any variable t. Thus setting $t = \frac{3\bar{r}}{\sqrt{D-\bar{d}}}\sqrt{2\left(d + \ln 2\right)}$ we have

$$\begin{aligned}\int_0^\infty \mathbb{P}[\rho \geq t]\mathrm{d}t &= \int_0^l \mathbb{P}[\rho \geq t]\mathrm{d}t \ + \ \int_l^\infty \mathbb{P}[\rho \geq t]\mathrm{d}t \\ &\leq \int_0^l 1\mathrm{d}t \ + \ \int_1^0 \eta\mathrm{d}\eta \\ &= \frac{\bar{r}\sqrt{D}}{2\Delta\sqrt{D-\bar{d}}}\left[\sqrt{2(d+\ln 2)} + \int_0^1 \frac{\mathrm{d}\eta}{\sqrt{2(d+\ln\frac{2}{\eta})}}\right]\end{aligned}$$

Looking at solving $\int_0^1 \frac{\mathrm{d}\eta}{\sqrt{2(d+\ln\frac{2}{\eta})}}$, we substitute the variable η simplifying things to solve

given its limits. Set $u = \sqrt{d + \ln \frac{2}{\eta}}$

$$\begin{aligned}
&\Rightarrow u^2 = d + \ln \frac{2}{\eta} \Rightarrow \mathrm{e}^{u^2} = \mathrm{e}^{d+\ln \frac{2}{\eta}} \\
&\Rightarrow \mathrm{e}^{u^2} = \mathrm{e}^d \frac{2}{\eta} \Rightarrow \eta = 2\mathrm{e}^d \mathrm{e}^{-u^2} \\
&\Rightarrow \mathrm{d}\eta = 2\mathrm{e}^d(-2u)\mathrm{e}^{-u^2}\mathrm{d}u
\end{aligned}$$

We substitute the limits so given $g(\eta) = \sqrt{2(d + \ln \frac{2}{\eta})}$
for $\eta = 0 \Rightarrow g(0) = \lim_{\eta \to 0}(d + \ln \frac{2}{\eta}) = \infty$ and $\eta = 1 \Rightarrow g(1) = \sqrt{d + \ln \frac{2}{1}} = \sqrt{d + \ln 2}$, continuing in providing the upper bound:

$$\leq \frac{\bar{r}}{2\Delta} \left[\sqrt{2(d + \ln 2)} + 2\sqrt{2}\mathrm{e}^d \int_{\sqrt{d+\ln 2}}^{\infty} \mathrm{e}^{-u^2}\mathrm{d}u \right]$$

An integral of the form $\int_a^{\infty} \mathrm{e}^{-u^2}\mathrm{d}u$ is solved in the following manner:

$$\begin{aligned}
\int_a^{\infty} \mathrm{e}^{-u^2}\mathrm{d}u &= \frac{1}{2}\left[\int_{-\infty}^{\infty} \mathrm{e}^{-u^2}\mathrm{d}u + \int_a^a \mathrm{e}^{-u^2}\mathrm{d}u \right] \\
&\leq \frac{\pi}{2}\left[1 - \sqrt{1 - \mathrm{e}^{-a^2}} \right] \leq \frac{\pi}{2}\mathrm{e}^{-a^2}
\end{aligned}$$

since its true that for $0 < x < 1 \Rightarrow 1 - \sqrt{1 - x} < x$. If we let $d \geq 1$ we arrive at the probability upper bound of ball B getting split $\frac{\bar{r}}{2\Delta}\left[\sqrt{2(d + \ln 2)} + \sqrt{\frac{\pi}{2}} \right] \leq \frac{3\bar{r}\sqrt{d}}{2\Delta}$. $\square$

3.4 Proof of Size Reduction Theorem 7

Consider two balls in the defined cover, $B = B(z, \Delta)$ and $B' = B(z', \Delta)$ where $z, z' \in B(x, \Delta)$ and $\|z - z'\| \geq \frac{\Delta}{2} - r$. A split is either a "good split", a "bad split" or a "neutral split". A "good split" is defined as one that separates B and B', a "bad split" as one that intersects both B and B' and a "neutral split" as one that intersects one or none of the two balls.

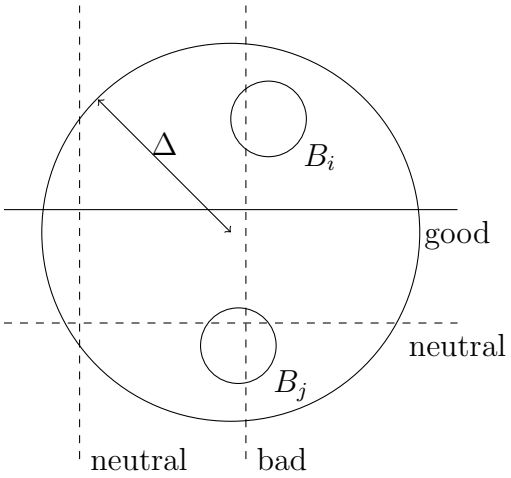


Figure 3-1: Depicted are the three types of splits in a 2-dim scenario. Good split cleanly separates $B(x, \Delta)$ and $B'(x, \Delta)$. Bad split intersects both B and B'. Neutral split either intersects just one ball, or keeps both balls on the same side of the split.

Lemma 16. *Let $S \subset B(x, \Delta)$ have doubling dimension d. Pick balls $B = B(z, \Delta)$ and $B = B(z', \Delta)$ and a space V of dimension D such that*

1. $z, z' \in B(x, \Delta)$
2. $\|z - z'\| \geq \frac{\Delta}{2} - r$
3. $r \leq \frac{\Delta}{(288\sqrt{d})}$

Let v be a random unit vector orthogonal to V, and s be a point uniformly at random in the interval $[median - 6\Delta/\sqrt{D},\ median + 6\Delta/\sqrt{D}]$. Then with probability at least $\frac{1}{96 \cdot s}$, $\pi_v(B)$ and $\pi_v(B')$ lie on different sides of s.

Proof. For $\delta = 2/e^{31}$ and $r \leq \Delta/288\sqrt{d}$, then $\pi_v(B)$ is within the interval of radius $3\frac{r}{s \cdot \sqrt{D - \bar{d}}}\sqrt{2(d + \ln(2/\delta))}$:

$$\begin{aligned} 3\frac{r}{s \cdot \sqrt{D - \bar{d}}}\sqrt{2(d + \log(2/\delta))} &\leq \frac{3\Delta}{288 \cdot s \cdot \sqrt{d\left(D - \bar{d}\right)}}\sqrt{2\log 2(e(2/\delta))} \\ &\leq \frac{\Delta}{96 \cdot s}\sqrt{\frac{64}{D - \bar{d}}} \\ &\leq \frac{\Delta}{12 \cdot s \cdot \sqrt{D - \bar{d}}} \end{aligned}$$

We also show that the projected distance between the small ball centers is:

$$\|\pi_v(z - z')\| \geq \frac{\Delta}{4 \cdot s \cdot \sqrt{D - \bar{d}}}$$

Applying Lemma 9 (a) and setting $\alpha = \sqrt{\frac{10}{9}}/4$

$$\mathbb{P}\left(\|\pi_v \pi_W(z - z')\| \leq \alpha \frac{\|\pi_W(z - z')\|}{s \cdot \sqrt{D - \bar{d}}} \right) \leq \alpha \sqrt{\frac{2}{\pi}}$$

$$\mathbb{P}\left(\|\pi_v \pi_W(z - z')\| \leq \alpha \sqrt{9/10} \frac{\|z - z'\|}{s \cdot \sqrt{D - \bar{d}}} \right) \leq \alpha \sqrt{\frac{2}{\pi}}$$

$$\mathbb{P}\left(\|\pi_v \pi_W(z - z')\| \leq \frac{\Delta}{4 \cdot s \cdot \sqrt{D - \bar{d}}} \right) \leq \frac{1}{4} \sqrt{\frac{20}{9 \cdot \pi}} < \frac{3}{14}$$

With probability at least $1 - \frac{3}{14}$, there is a gap of at least between $\pi_v(B)$ and $\pi_v(B')$, given by

$$\frac{\Delta}{4 \cdot s \cdot \sqrt{D - \bar{d}}} - 2 \frac{\Delta}{12 \cdot s \cdot \sqrt{D - \bar{d}}} \geq \frac{\Delta}{4 \cdot s \cdot \sqrt{D}}$$

$$\begin{aligned}
&\mathbb{P}[\text{B and B' cleanly separated}] \\
&\geq \mathbb{P}[\text{U is good}] \cdot \mathbb{P}[\text{B and B' cleanly separated} \mid \text{U is good}] \\
&\geq \frac{1}{2} \cdot \frac{\Delta/4 \cdot s \cdot \sqrt{D}}{12\Delta/\sqrt{D}} = \frac{1}{96 \cdot s}
\end{aligned}$$

As stated a clean separation is expected with probability at least $\frac{1}{96 \cdot s}$. □

Lemma 17. *Given the hypothesis of Lemma 16,*

$$\mathbb{P}[\pi_v(B),\ \pi_v(B') \textit{ both intersect the split point }] \ < \ \frac{1}{192 \cdot s}$$

Proof. Requiring a bound inversely proportional to s (as noted in lemma 11 of [12]) the probability that a bad split is less than $\min\{\mathbb{P}(E_{B_1}), \mathbb{P}(E_{B_2})\}$, where E_B is the

event that ball B is split. Thus, noting the previous observation and applying the result of lemma 16 to it, a random split of the cell is a bad split with respect to this pair with probability $\leq \frac{1}{192 \cdot s} \left(= \frac{3\bar{r}\sqrt{d}}{2\Delta} \cdot \frac{\Delta}{288 \cdot s \cdot \sqrt{d}} \right)$. □

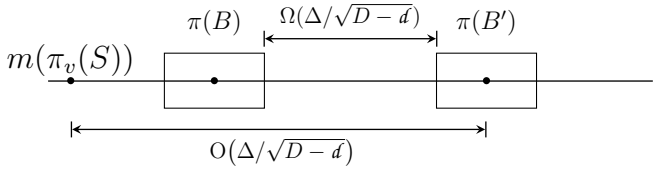


Figure 3-2: Bounded distance of each ball from each other and from projected median m

Setting up the proof for Theorem 7, starting in a cell C containing a pair of balls the probability that a cell k levels below levels data from both the balls is exponentially small in k. Considering the traversal of enough levels, we can take a union bound over all pairs of balls whose centers are well separated thus resulting in the desired proof of the Theorem.

Proof of Theorem 7. Let $S \subseteq B(x, \Delta)$ with doubling dimension d, be a set of points in a cell C of the tree with a diameter $\Delta = \text{diam}(C \cap S)$. S can be covered using $O\big((sd)^{2d}\big)$ balls, with radii $\Delta/288 \cdot s \cdot \sqrt{d}$, where the centers are separated by at least $\Delta/s - \Delta/288s\sqrt{d}$. Let $\bar{d} < c_0 D$ and $\{v_1, \ldots, v_D\}$ be a set of random orthonormal vectors, with $W = (v_1, \ldots, v_{\bar{d}})^{\perp}$. Before we continue any further, we prove the following:

Lemma 18. *Let p_i^j be the probability that cell i levels below C has a descendant j levels further below cell i which contains data from both balls previously defined to lie in C. We want to show that $p_k^0 \leq \left(1 - \frac{1}{192s}\right)^l \cdot p_{k-l}^l$.*

Proof.

$$
\begin{aligned}
p_k^0 \;\leq\; & \mathbb{P}\left[\text{“good split” at level } 0\right] \cdot 0 + \\
& \mathbb{P}\left[\text{“bad split” at level } 0\right] \cdot 2p_{k-1}^1 + \\
& \mathbb{P}\left[\text{“neutral split” at level } 0\right] \cdot p_{k-1}^1 \\
& \leq \frac{1}{192s} \cdot 2p_{k-1}^1 + \left(1 - \frac{1}{192s} - \frac{1}{96s}\right) \cdot p_{k-1}^1 \\
& \leq \left(1 - \frac{1}{192s}\right)^2 \cdot p_{k-2}^2 \\
& \vdots \\
& \leq \left(1 - \frac{1}{192s}\right)^l \cdot p_{k-l}^l
\end{aligned}
$$

□

An inductive proof remains to show the final result that $L(s) = \mathrm{O}\big(d \log s \log sd\big)$. Let $s = 2$ then by Theorem 5 we have a “good split” with probability at least $\frac{1}{192}$ and “bad split” at most $\frac{1}{384}$. For the inductive step we assume that with probability $> \; 1 - \frac{1}{4}$, in $L(s)$ levels, the size if all the descendants goes down by a logarithmic factor s. Set the probabilities of a “good split” and a “bad split” in a cell at a depth l with notation p_g^l and p_b^l respectively. Assume that E the event that the radius of each cell at level $\bar{l} = L(s/2)$ such that $\bar{l} < \frac{\Delta}{s/2}$, where cell is $\bar{C}$.

$$
\begin{aligned}
p_g^{\bar{l}} \;\geq\; & \mathbb{P}\left[\text{“good split” in } \bar{C} | E\right] \cdot \mathbb{P}\left[E\right] \geq \frac{1}{196} \cdot \left(1 - \frac{1}{4}\right) \geq \frac{1}{296} \\
p_b^{\bar{l}} \;=\; & \mathbb{P}\left[\text{“bad split” in } \bar{C} | E\right] \cdot \mathbb{P}\left[E\right] + \mathbb{P}\left[\text{“bad split” in } \bar{C} | \neg E\right] \cdot \mathbb{P}\left[\neg E\right] \\
\leq & \frac{1}{384} \cdot 1 + \frac{1}{384} \cdot \frac{1}{4} \leq \frac{1}{306}
\end{aligned}
$$

For any $m > 0$, $p_m^{\bar{l}} \leq \left(1 - \frac{1}{9057}\right)$ For constant c_4 and $k = \bar{l} + c_4 d \log sd$ and applying lemma 18, $p_k^0 \leq \left(1 - \frac{1}{192s}\right)^{\bar{l}} \cdot \left(1 - \frac{1}{9057}\right)^{c_4 d \log sd} \leq \frac{1}{4(sd)^{2d}}$.

Solving the recurrence,

$$\begin{aligned} L(s) &\leq L(s/2) + c_4 d \log sd \\ &\leq L(s/2^2) + 2c_4 d \log sd \\ &\leq L(s/2^2) + 3c_4 d \log sd \\ &\cdots \\ &\leq L(s/2^i) + ic_4 d \log sd \end{aligned}$$

For $i = \log s \Rightarrow L(s) = L(s/2^{\log s}) + c_4 d \log s \log sd$. Thus, $L(s) = \mathrm{O}\big(d \log s \log sd\big)$.

□

Chapter 4

Nearest Neighbor Algorithm

4.1 Structure Properties

Prior to describing the approximate nearest neighbor query algorithm and showing the search time we need to define the properties of the Randomly-oriented RKD-trees structure which relate to ANN query search. The verification of these properties is useful when providing an upper bound to the number of leaf cells visited by approximate nearest (and k-nearest) neighbor queries and ultimately providing an upper bound on the a query's ANN (k-ANN) search time.

Given a query point $q \in \mathbb{R}^D$, a simple descent of the tree assures that the cell associated with q will be visited in $\mathrm{O}\big(\log N\big)$ time. The height of the tree is $\log N$, and a $\mathrm{O}\big(1\big)$ comparison of the cut-dimension coordinate is performed at each level down to a leaf. In addition, each leaf cell visited is associated with exactly one point. These two properties are a consequence of the construction of Randomly-oriented RKD-trees.

The next significant property we address is the roundness of the cells or the *aspect ratio*, that is, the ratio between the shortest and longest length of any of the cells. Having very elongated cells to search through in a tree may increase query times up to $\mathrm{O}\big(N\big)$. An assurance that the cells created have an aspect ratio that does not fall under small thresholds, no matter how unaccommodating the clustering of points is, may be of vital importance to the search time.

A property which relates to backtracking efficiency of a tree, is that the number of cells of size at least r that intersect an open ball of radius $R > 0$ be bounded by above by some function. The packing guarantee is later show to be the multiplicative factor of the search time. The aspect ratio bound is used along with the result of Theorem 7 to prove that a packing guarantee exists. We are fortunate since we are able to use similar arguments to the ones used in [12] to prove bounds for a The RPTree-Max.

The proof is a two step process. First we show that the proposed structure will, with high probability, inscribe any ball B, as defined in the above theorem, in a Randomly-oriented RKD-tree cell C with radius no more than $\mathrm{O}\big(Rd\sqrt{d}\log d\big)$. In the second part we show that the number of disjoint cells of radius at least r that intersect ball B is bounded by the number of descendants of C with radius r. An upper bound is achieved with some help from theorem 6.

We aim to prove an upper bound on the radius of the smallest Randomly-oriented RKD-tree cell that completely contains a given ball B of radius R. This is what bounds the aspect ratio of this cell. Let there be balls of radius $\Delta/512\sqrt{d}$ surrounding B at a distance of $\Delta/2$ (see Figure 4-1), covering the annulus centered at B of mean radius $\Delta/2$ and thickness $\Delta/512\sqrt{d}$. The annulus may be covered by $d^{\mathrm{O}\left(d\right)}$ balls. This fact suffices in order to make our point. Without loss of generality assume that the centers of these balls lie in C.

Notice that if B gets separated from the balls, without getting split in the process, then it will lie in a cell of radius $< \Delta/2$. Fix a ball B_i, we define two types of random split of the Randomly-oriented RKD-tree: (a) *useful* if it separates B from B_i, and (b) *useless* if it splits B. We use two facts, first slightly manipulating lemma 16, we have the probability of a useful split being $\frac{1}{96\cdot s}$, second Lemma 15 which says that the probability of a useless split is at most $\frac{3R\sqrt{d}}{2\Delta}$.

Lemma 19 (taken from Lemma 14 of [12])**.** *There exists a constant c_5 such that the probability of a ball of radius R in a cell of radius Δ getting split before it lands up in a cell of radius $\Delta/2$ is at most $\frac{c_5 Rd\sqrt{d}\log d}{\Delta}$.*

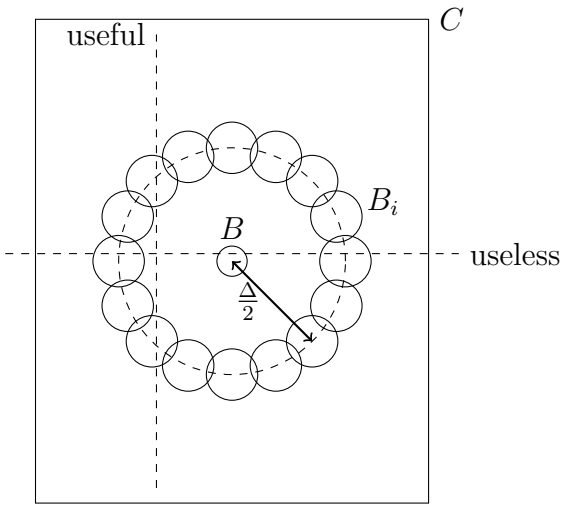


Figure 4-1: The centers of each ball B_i is located at a distance $\Delta/2$ from the center. Their radius is $\Delta/512\sqrt{d}$

Proof. (Adapted from proof of lemma 14 [12].)

$$\mathbb{P}[E[k]] \leq min\left\{1, \left(1 - \frac{1}{192}\right)^{k-1} \cdot d^{\mathrm{O}(d)}\right\} \cdot \frac{3R\sqrt{d}}{2\Delta}$$

It is true that $\mathbb{P}[\mathbb{E}] \leq \sum_{k>0} \mathbb{P}[E[k]]$, thus:

$$\begin{aligned}
\mathbb{P}[\mathbb{E}] &\leq \sum_{k>0} \mathbb{P}[E[k]] \\
&= \sum_{k>0} min\left\{1, \left(1 - \frac{1}{192}\right)^{k-1} \cdot d^{\mathrm{O}(d)}\right\} \cdot \frac{3R\sqrt{d}}{2\Delta} \\
&\leq \left(\sum_{i=1}^{n} 1 + \sum_{i=1}^{\infty} \frac{1}{4}\left(1 - \frac{1}{192}\right)^{k} \cdot d^{\mathrm{O}(d)}\right) \cdot \frac{3R\sqrt{d}}{2\Delta}
\end{aligned}$$

Given we may assume $n = \mathrm{O}\big(d \log d\big)$ the probability of event E is bounded by $\mathrm{O}\big(\frac{Rd\sqrt{d}\log d}{\Delta}\big)$. □

Lemma 20 (taken from Lemma 15 of [12])**.** *There exists a constant c_6 such that with probability $> 1 - 1/4$, a given ball B of radius R will be completely inscribed in an Randomly-oriented RKD-tree cell C of radius no more than $c_6 \cdot Rd\sqrt{d}\log d$.*

Proof. (Taken from the proof of Theorem 15 [12]) Let $\Delta^* = 4c_5 Rd\sqrt{d}\log d$ and $\Delta_{\max}$ be the radius of the entire dataset. The event that ball B is unsplit in a cell of radius $\frac{\Delta_{\max}}{2^i}$ may be denoted as $F[i]$. $\mathbb{P}[F[m]|F[m-1]] = \mathbb{P}[E]$, where $\mathbb{P}[E]$ has been defined previously in Lemma 19 and $m = \log\frac{\Delta_{\max}}{\Delta^*}$. Trivially $\mathbb{P}[F[m]|\neg F[m-1]] = 0$. The probability $\mathbb{P}[F[m]]$ is then:

$$\begin{aligned}\mathbb{P}[\mathbb{E}] &= \prod_{i=0}^{m-1}\mathbb{P}[F[i+1]|F[i]] = \prod_{i=0}^{m-1}\left(1 - \frac{c_5 Rd\sqrt{d}\log d}{\Delta_{\max/2^i}}\right)\\ &\geq 1 - \sum_{i=0}^{m-1}\frac{c_5 Rd\sqrt{d}\log d}{\Delta_{max}/2^i} = 1 - \sum_{i=0}^{m-1}\frac{c_5 Rd\sqrt{d}\log d}{2^{m-i}\Delta^*}\end{aligned}$$

Let $c_6 = 4c_5$,

$$= \sum_{i=0}^{m-1}\frac{c_5 Rd\sqrt{d}\log d}{2^{m-i}4c_5 Rd\sqrt{d}\log d} = 1 - \frac{1}{4}\sum_{i=0}^{m-1}\frac{1}{2^{m-i}} \geq 1 - \frac{1}{4}$$

□

Theorem 21 (taken from Theorem 13 of [12])**.** *Given a fixed ball $B(x,R) \subset \mathbb{R}^D$, with probability greater than $1/2$ (where the randomization is over the construction of the Randomly-oriented RKD-trees), the number of disjoint Randomly-oriented RKD-trees cells of radius greater than r that intersect B is at most $\left(\frac{R}{r}\right)^{\mathrm{O}\left(d\log d\log(dR/r)\right)}$.*

Proof. (of Theorem 21, taken from proof of Theorem 13 [12]) Given we have a ball B of radius R that lies in a cell C then Theorem 19 shows that with probability at least $3/4$ cell C will have a radius of at most $R' = \mathrm{O}\left(Rd\sqrt{d}\log d\right)$. We need to establish an upper bound on the number of disjoint cells of radius at least r, this can be done by counting the number of descendants of C of radius no less than r. From Theorem 6 we know that with probability at least $3/4$ the radius of the cells will decrease to a radius below r in $\log(R'/r)d\log(R'/r)$ levels of reduction. Thus, the number of children is at most $2^{(\log(R'/r)d\log(R'/r))}$ with a success probability of at most $3/4 \cdot 3/4 = 9/16 > 1/2$. □

Now that we have established a packing guarantee, what remains is to bound

the time it takes to prioritize the cells around the query point. The method is a more complex version of the *priority search* techniques used for kd-trees by Arya and Mount [3]. The added complexity arises from the fact that multiple near cells are prioritized from T different trees.

Given that T the number of Randomly-oriented RKD-trees built in the construction process, the search algorithm begins its process by searching through each tree one at a time. Beginning with the root of the tree we recursively repeat the same process. We extract the cut-dimension and cut-value associated with the node. The query's cut-dimension coordinate value is compared to cut-value, consequently the next node traversed (left or right) is determined. The branch not visited is added to a priority heap; the heap is sorted according to the accumulated L_1 distance, $accumDist = accumDist + (q[cutDimension] - cutValue)$, along the path so far. Each leaf node visited, is added to a set of visited leafs. If a leaf has previously been visited in another tree then it is not visited again. If it has not, then it is inserted along with its Euclidean distance to the query point and index value into the priority queue. An added parameter also keeps track of the number of leafs visited, if it exceeds a predefined value then the function terminates. The proof of correctness is similar to the proof provided by Arya and Mount [3], being mindful of the fact that a search of multiple trees may visit the same points multiple times.

There are at most $O(N)$ nodes within a heap, thus the minimum may be extracted in $\mathrm{O}\big(\log N\big)$ time. Each step of the tree descent is processed in $\mathrm{O}\big(1\big)$ time plus $\mathrm{O}\big(1\big)$ to insert a node into the heap, assuming we use a Fibonacci heap [16]. Thus the time needed to enumerate the nearest m cells to the query point is $O(m \log N)$

4.2 Approximate Nearest Neighbor Queries

We now have the capacity to show what the time needed to answer a $(1 + \epsilon)$-approximate nearest neighbor (Theorem 2), given that the spatial subdivision occurs with a data structure that satisfies the properties highlighted in the previous section. The function depends on the dimension D, the error ϵ and the data set size N. Given

a query point $q \in \mathbb{R}^D$, the output of the algorithm is a point p whose distance from the query point, $dist(q,p)$ is at most a factor $(1+\epsilon)$ greater than the distance of the query point from the true nearest neighbor p_{nn}.

Recall that the algorithm induces randomization in the formation of T different subdivisions of the same space. The algorithm begins by traversing each of the T trees, down to a leaf cell containing exactly one data point. The randomized quality of the algorithm increases the probability that the point residing in the cell containing the query point will differ for two more of the T different space subdivisions. The ANN candidate is one of the T points examined that is nearest to the query point.

In the process of each traversal of the trees, the algorithm maintains a sorted heap (sorting by distance to the query point) that contains the branches not taken in the traversal. A branch is added if its L_1 distance from the query point is less that a $1+\epsilon$ factor from the current ANN candidate. If a branch is traversed and the leaf cell visited contains a closer point to the query than the current ANN candidate. The search terminates and returns the approximate nearest neighbor when all the distances of the branches in the heap are greater than the distance to the ANN candidate.

The total query time is established with the help of the following lemma:

Lemma 22. *The number of leaf cells examined by the ANN algorithm is at most* $\left(\frac{d}{\epsilon}\right)^{O\left(d \log d \log\left(d\frac{D}{\epsilon}\right)\right)}$ *for the* L_1 *metric.*

Proof. If the distance from the query point to the last leaf cell checked is denoted as l. We know that all the cells traversed so far are within the distance l from the query point. If p is the candidate approximate nearest neighbor thus far, then $l(1+\epsilon) \leq dist(q,p)$.

We claim that no cell seen so far can be of size less than $l\epsilon/D$. To prove by contradiction we suppose that a cell of size greater than $l\epsilon/D$ has in fact been visited. The cell is within distance l of q, and hence overlaps a ball of radius l centered at q. The diameter of the cell in the L_1 metric is at most D times the longest length, thus is less than $D \cdot \frac{l\epsilon}{D} = l\epsilon$. The search must have encountered a point at a distance

less than $l + l\epsilon = l(1+\epsilon)$ from q. However the result contradicts that point p is the current candidate approximate nearest neighbor thus far.

The number of cells visited up until the algorithm returns is bounded by the number of cells of size at least $l\epsilon/D$ than can overlap a ball of radius l centered at q. From the packing lemma we have shown earlier and the fact stated in this proof, the number of cells is at most $\left(\frac{R}{R\epsilon/D}\right)^{O\left(d \log d \log (d\frac{R}{R\epsilon/D})\right)} = \left(\frac{D}{\epsilon}\right)^{O\left(d \log d \log (d\frac{D}{\epsilon})\right)}$. □

The result of lemma 22 is used to show Theorem 2.

Chapter 5

Implementation

Given a set of N data points in D-dimensional space, queries are performed for the given inputs (a) query points: $\{q_1, \ldots q_l\}$, (b) number of nearest neighbors nn, (c) error ϵ, (d) search nearest/furthest (*true/false*) and (e) perform exact or approximate search (*true/false*). The points in Euclidean space of dimension D are identified with tuples of D real numbers, with the Cartesian product $\mathbb{R}^D$. They are preprocessed into a space partitioning data structure, such that, given any query item q_i the set of points can be browsed efficiently. The *Randomly-oriented RKD-trees* spatial searching package is designed for data sets that are represented in medium to large dimension D with the objective of performing highly time-efficient approximate spatial queries in pursuit of one or more approximate nearest neighbors. The space is intended to be small enough to store the search structure in main memory.

The package contains the implementation written in *C++*. It is written using the functions *Computational Geometry Algorithms Library* (CGAL). It is meant to extend the current functionality offered in the *dD Spatial Searching*, increasing search time efficiency for medium and large dimensional datasets.

5.1 Build

Provided the dataset D-dimensional points the build process is performed as a two tear process (1) rotation of the ambient space, (2) creation of t trees, the combination

of which defines the data structure.

5.1.1 Random rotation

Looking initially at the 2-D case, we gain some insight on how the random rotation matrix is generated in higher dimensions. Generate a uniformly distributed random rotation matrix R. The rotation angle θ is uniformly distributed between 0 and 2π. Let

$$\theta = 2\pi \cdot \mathcal{U}(0,1) \text{ and set } R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix},$$

the countercolockwise rotation of a vector v by an angle θ. Obtain a rotated data set $S' = R \times S$ for data set S.

Definition 23 (Orthogonal group)**.** [1]

$$O(n) = \{M \in \mathrm{Mat}(n, \mathbb{R}) : M'M = I\}$$

$\mathrm{Mat}(n, \mathbb{R})$ *denotes the space of all* $n \times n$ *real matrices; and* M' *denotes the transpose of matrix* M

Definition 24 (Special orthogonal group)**.** [2] *The special orthogonal group*

$$SO(n) = \{M \in O(n) : \det M = 1\}$$

is a closed group of the compact group $O(n)$

$SO(n)$ is itself compact. Note that

$$M \in O(n) \Rightarrow \det M = \pm 1$$

since

$$M'M = I \Rightarrow \det M' \det M = 1 \Rightarrow (\det M)^2 = 1$$

[1] http://www.maths.tcd.ie/pub/coursework/424/GpReps-II.pdf
[2] http://www.maths.tcd.ie/pub/coursework/424/GpReps-II.pdf

Algorithm 1: Random Rotation

input : The dimension d, the dataset (matrix) R
output: Rotated dataset (matrix)

```
if d = 0 then
    R.resize(0, 0);
    return;
end
if d = 1 then
    R.resize(1, 1);
    return;
end
random_theta ← 2* PI * uniform_number(0, 1);
cos_theta ← cos(random_theta);
sin_theta ← sin(random_theta);
R (0,0) = cos_theta;
R (0,1) = -sin_theta;
R (1,0) = sin_theta;
R (1,1) = cos_theta;
for i ← 0 to 1 do
    for j ← 0 to 1 do
        R_submat(i, j) ← R(i, j);
    end
end
A.zeros(d,d);
for dmc ← 2 to d do
    v_vector ← random_unit_vec(dmc);
    normalize(v_vector);
    R_submat.resize(dmc, dmc);
    R_submat(dmc, dmc)← 1;
    R_submat_orthonormal ← graham_schmidt(R_submat);
    for i ← 0 to dmc do
        for j ← 0 to dmc do
            R(i, j) ← R_submat(i, j);
        end
    end
end
if determinant(R) < 0 then
    R.col(0) ← -R.col(0)
end
return R;
```

since $\det M^{'} = \det M$.

In high dimensions, using the subgroup algorithm of Diaconis & Shashahani (1987) [13], we recursively exploit the nested dimensions group structure of $SO(n)$. This is done as follows. Generate a uniform angle θ and construct a 2×2 rotation matrix R. Stepping from n to $n+1$, generate a vector v uniformly distributed on the n-sphere, aided by the GramSchmidt process[3] [33]. S_n embed the $n \times n$ matrix in the next larger size with last column $(0, \ldots, 0, 1)$ and rotate the larger matrix so the last column becomes v.

GramSchmidt process

Define the projection operator by

$$proj_{\mathbf{u}}(\mathbf{v}) = \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\langle \mathbf{u}, \mathbf{u} \rangle} \cdot \mathbf{u}$$

where $\langle \mathbf{u}, \mathbf{v} \rangle$ denotes the inner product of vectors $\mathbf{u}$, $\mathbf{v}$. The operator projects the vector $\mathbf{v}$ into the line spanned by vector $\mathbf{u}$. For $\mathbf{u} = 0$ define $\mathrm{proj}_0(\mathbf{v}) := 0$. Then the Graham-Schmidt process works as follows:

$$\begin{aligned}
\mathbf{u}_1 &= \mathbf{v}_1, & \mathbf{e}_1 &= \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|} \\
\mathbf{u}_2 &= \mathbf{v}_2 - \mathrm{proj}_{\mathbf{u}_1}(\mathbf{v}_2), & \mathbf{e}_2 &= \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} \\
\mathbf{u}_3 &= \mathbf{v}_3 - \mathrm{proj}_{\mathbf{u}_1}(\mathbf{v}_3) - \mathrm{proj}_{\mathbf{u}_2}(\mathbf{v}_3), & \mathbf{e}_3 &= \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|} \\
\mathbf{u}_4 &= \mathbf{v}_4 - \mathrm{proj}_{\mathbf{u}_1}(\mathbf{v}_4) - \mathrm{proj}_{\mathbf{u}_2}(\mathbf{v}_4) - \mathrm{proj}_{\mathbf{u}_3}(\mathbf{v}_4), & \mathbf{e}_4 &= \frac{\mathbf{u}_4}{\|\mathbf{u}_4\|} \\
&\vdots & &\vdots \\
\mathbf{u}_k &= \mathbf{v}_k - \sum_{j=1}^{k-1} \mathrm{proj}_{\mathbf{u}_j}(\mathbf{v}_k), & \mathbf{e}_k &= \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}.
\end{aligned}$$

The normalized vectors $\mathrm{e}_1, \ldots, \mathrm{e}_k$ form an orthonormal set.

[3] http://en.wikipedia.org/wiki/Gram-Schmidt_process

We implement the above algorithm based on the MATLAB implementation titled: "Simplex and random rotation", Maxim Vedenyov (2012) [4]

Algorithm 2: Build

input : The data set data, the data set size size, the data's dimension d, the bucket size bucket_size, number of trees number_of_trees

```
points ← matrix_multiply(data,random_rotation(data));
// For each tree save a root
for j ← 1 to number_of_trees do
    for i ← 0 to size do
        index [i] ← i;
    end
    // Uniform random shuffle the index array
    index ← shuffle(index);
    if size ≤ bucket_size then
        tree_root [j] ← leaf_node(points,size,d,bucket_size);
    end
    else
        tree_root [j] ← internal_node(points,size,d,bucket_size);
    end
end
```

5.1.2 Hyperplane splits

The rule defined by the mean split estimates the mean and variance of each $1 \ldots D$ dimensions of the dataset. A hyperplane is used to separate the half spaces in each node. They are defined by a *cutting-dimension* (cd) and *cutting-value* (cv) [5]. The median split is similar to that above, difference being the fact that a random point is chosen in a closed range near the true median.

The variance of dimensions with the highest values are pooled together out of which one is chosen uniformly at random. The dimension chosen in the previous step is set to be the cutting-dimension associated with the node. The cutting-value is equal to the mean at of the cut-dimension.

[4] http://www.mathworks.com/matlabcentral/fileexchange/38187-simplex-and-random-rotation/content/random_rotation.m

[5] http://doc.cgal.org/latest/Spatial_searching/classCGAL_1_1Plane__separator.html

Algorithm 3: InternalNode

input : The node points points, the data set size size, the data's dimension d, the bucket size bucket_size

output: The node node

```
cd ← cut_dimension(points, size, d);
cv ← cut_value(points [cd ]);
foreach point in points do
    if point ≤ cv then
        left_points.add(point);
    end
    else
        right_points.add(point);
    end
end
if left_points.size() ≤ bucket_size then
    node.left_child ← leaf_node(left_points, left_points.size(), d,
    bucket_size);
end
else
    node.left_child ← internal_node(left_points, left_points.size(), d,
    bucket_size);
end
if right_points.size() ≤ bucket_size then
    node.right_child ← leaf_node(right_points, right_points.size(), d,
    bucket_size);
end
else
    node.right_child ← internal_node(right_points,
    right_points.size(), d, bucket_size);
end
return node
```

Algorithm 4: LeafNode

input : The node points points, the data set size size, the data's dimension d, the bucket size bucket_size

output: The node node

```
// Each point in the bucket size.  The size is commonly equal
   to one
foreach point in points do
    node.data ← point;
end
return node;
```

5.2 Search

For each query point a search is performed. A sorted queue is saved with size equal to the number of nearest neighbors expected to return. The queue inputs are sorted according to the Euclidean distance from the query point. The point traverses each tree down to a leaf node containing a single point (or equal to a predefined bucket size). If the queue is not full then a point is pushed, otherwise, points are placed into the sorted queue if the distance is less than the maximum distance in the queue. As the query traverses the tree a record of the branches not taken is kept in a sorted heap. The sorted heap contains branches from all t trees. The branches may contain nearer neighbors than the ones in the priority queue. This is shown as an experimentally efficient backtracking method [29].

Once the first traversal is performed on each tree the sorted heap is 'popped' one by one and the branch is traversed down to a leaf node referencing a point. This process is performed until the maximum number of allowed searches is performed or until the error factor specifies that no more searches need to be performed and the resulting queue can be returned as output.

The best matches (stored in the sorted queue) are a $(1 + \epsilon)$ approximation of the true nearest neighbor and are returned as output.

Algorithm 5: Search

input : The Node N, the branch distance branch_distance, the leaf count leaf_count, the max count max_count, the heap `heap`, leaf been previously visited `checked`, nearest neighbor queue `priority_queue`

```
if then
    return;
end
if N.is_leaf() then
    if checked(N.index()) or (leaf_count ≥ max_count ) then
        return;
        checked.add(N.index());
        leaf_count ++;
        priority_queue.add(N.index(),
                distance(N, query));
    end
end
split_data ← query.get_coord(cut_dim) - cut_value;
if split_data ≤ 0 then
    best_child ← N.left_node();
    other_child ← N.right_node();
end
else
    best_child ← N.right_node();
    other_child ← N.left_node();
end
branch_distance ← branch_distance + split_data;
if branch_distance ∗(1 + ϵ) <
                priority_queue(priority_queue.size()−1) then
    heap.insert(other_child, branch_distance);
end
search(best_child, branch_distance, leaf_count, max_count, heap,
checked);
```

Chapter 6

Experimental Results

The following results arise from queries performed on SIFT [24] of 128-dimension, 350K size and Gisette [19] of 5000-dimension, 13.5K size datasets performed on a Intel Core i7-2670QM CPU @ 2.20GHz 8 chip-set, with 5.7 GiB of memory. Each iteration is a query performed 100 times searching for the 20 nearest neighbors to each query point.

6.1 build-time

We begin by presenting the results of the build process. It is evident in Figure 6-1 that the build-time of CGAL k-d tree exceeds build-time of all other structures (see Table 6.1 for the list of structures tested). The CGAL k-d tree build-time is approximately eight times as slow as that of Randomly-oriented RKD-trees.

Structure	Plot Legend	Package	Authors
Randomly-oriented RKD-trees	ro-rkd	-	-
BBD-tree	bbd_ann	ANN	S. Arya & D. Mount
K-d tree	kd_ann	ANN	S. Arya & D. Mount
CGAL k-d tree	kd_cgal	dD Spatial Searching-CGAL	H. Tangelder & A. Fabri
RKD-tree	rkd_flann	FLANN	M. Muja & D. Lowe

Table 6.1: The list of data structures compared to produce the empirical results

The build-time of Randomly-oriented RKD-trees constructed upon 4 trees outperforms the BBD-tree build-time approximately four times. The build-times of different sizes of Randomly-oriented RKD-trees offer a competitive option with respect to the compared structures.

The result of performing a random rotation of the basis by multiplying the point set by a random rotation matrix is added to the Randomly-oriented RKD-trees build-times. Despite the added time needed for Randomly-oriented RKD-trees, the build-time maintains its competitive edge.

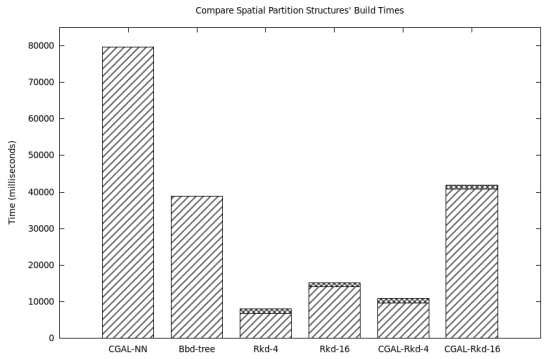


Figure 6-1: Both native and CGAL Randomly-oriented RKD-trees (4 trees) outperform all other structure build-times.

6.2 Query Time

The purpose of the following experiments is to perform an approximate k-nearest neighbor search query given a high-dimensional dataset and measure the search-time efficiency in relation to the precision. In this context precision is defined as the closeness in proximity to the exact k nearest neighbors when searching, with values ranging from 0 to 1, 1 being an exact match to the true k nearest neighbors of the given query set. A number of different structures (see Table 6.1) are tested against each other.

Both generated and 'real world' datasets are used. The 'real world' sets, a 128-dimensional SIFT [24] and the 5000-dimensional Gisette [19] dataset (see Table 6.3)

have been chosen for two reasons. One is to emphasize the efficiency displayed by Randomly-oriented RKD-trees when searching through datasets of medium and larger dimension size. Second, even more importantly, test the efficiency displayed by Randomly-oriented RKD-trees when an embedded low-intrinsic dimension lies within a much larger ambient dimension, the Gisette dataset contains low-intrinsic embedded dimension within the data's 5000 ambient dimension. The generated sets are a number of poison, uniform and a combination of both (see Appendix B). The aim is to test the data structures' search-time efficiency when the data may contain a few high variance dimensions in relation to the remaining ones.

Data Name:	SIFT-128		
Data Set Characteristics:	Multivariate	**Number of Instances:**	300K+
Attribute Characteristics:	Integer	**Number of Attributes:**	128

Table 6.2: SIFT dataset

The empirical running times performed on the SIFT data set show there is no significant deviation between the RKD-trees (FLANN) [26] algorithm and our Randomly-oriented RKD-trees when comparing times searched upon structures built with four and sixteen trees (6-2 (a), (b)). When a search is performed upon a sixteen tree structure the performance begins to deviate from four trees at values approximating 0.9 precision, given that 1 represents the true k-nearest neighbors. One of the more significant results of the paper is exhibited in 6-2 (d). The running times of the native k-d tree (CGAL) [31] is outperformed significantly throughout the domain of precision, up to an order of magnitude for the SIFT data.

The Randomly-oriented RKD-trees outperform the BBD-tree and k-d tree (ANN) [4] structures, though admittedly by a smaller margin than exhibited against the k-d tree (CGAL) structure. Another important observation (see Figure 6-3 (d)) is that the graphs slope stays relatively constant throughout the increasing precision values. A log scale plot (see Figure 6-3 (c)) gives a little clearer perspective regarding the margin

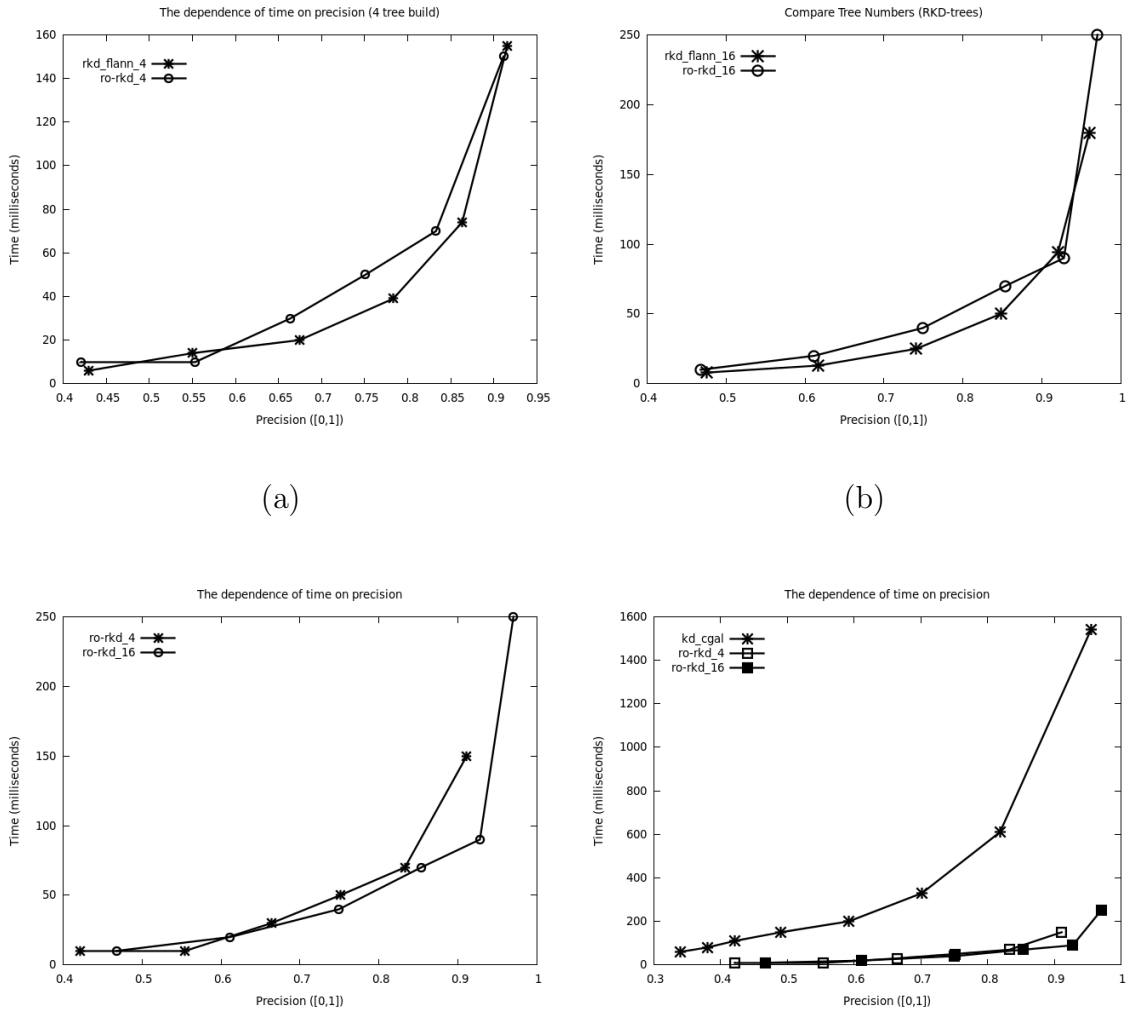
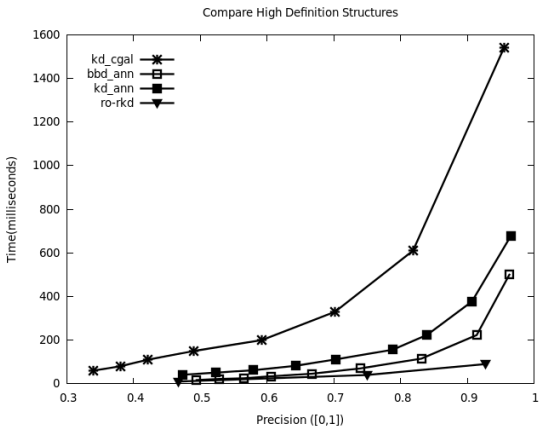
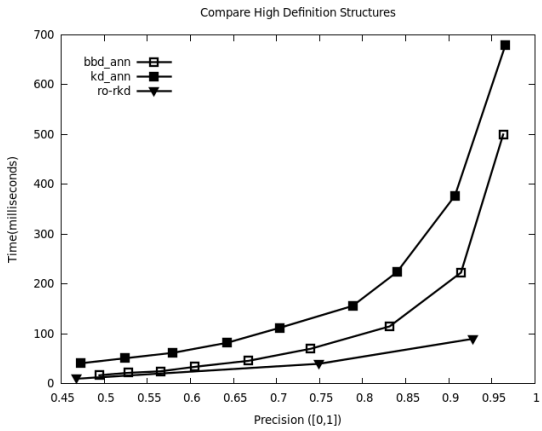


Figure 6-2: In (a) and (b) the graphs show the average searching time when searching for the 20 nearest neighbors in 100 different queries for the SIFT dataset for two different data structures, the Randomly-oriented RKD-trees structure implemented in CGAL and the RKD-trees of the FLANN package, for 4 and 16 trees respectively. Both implementations perform the queries in approximately the same time for all values of precision. (c) When comparing the same structure but using a different number of trees for the same task as that performed in figures (a) and (b) the graph shows that there the 16 trees outperform the 4 trees structure for high precision values. (d) The native k-d tree structure (CGAL) is outperformed by both 4 and 16 trees throughout the domain of precision.

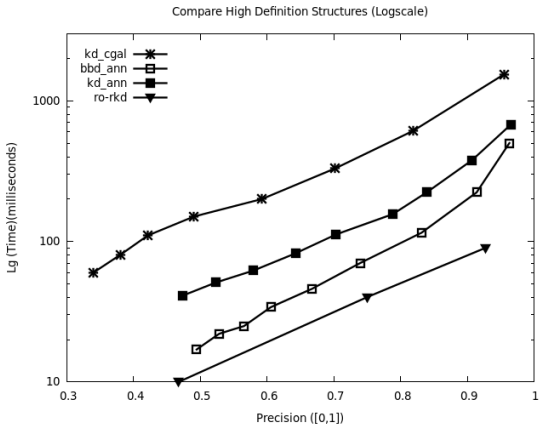
of difference, especially for less precise values, where it becomes difficult to observe the gap between the plotted lines.



(a)



(b)



(c)

Figure 6-3: (a) The k-d tree structure (CGAL) is outperformed by k-d tree and BBD-tree structures of the ANN package. When searching on the SIFT dataset, all three structures are outperformed by Randomly-oriented RKD-trees. (b) A more refined view of the k-d tree, BBD-tree and Randomly-oriented RKD-trees shows the difference in search-time efficiency. (c) The logarithmic scaled time shows that for values of less precision the trees are all outperformed by the Randomly-oriented RKD-trees throughout the precision domain.

The last set of running times (see Figure 6-4) are those displayed by the approximate nearest neighbor search algorithms' execution on the GISETTE data set. We observe the different structures' dependence on dimension size, additionally, we examine how the structures perform when the data set contains a lower dimensional

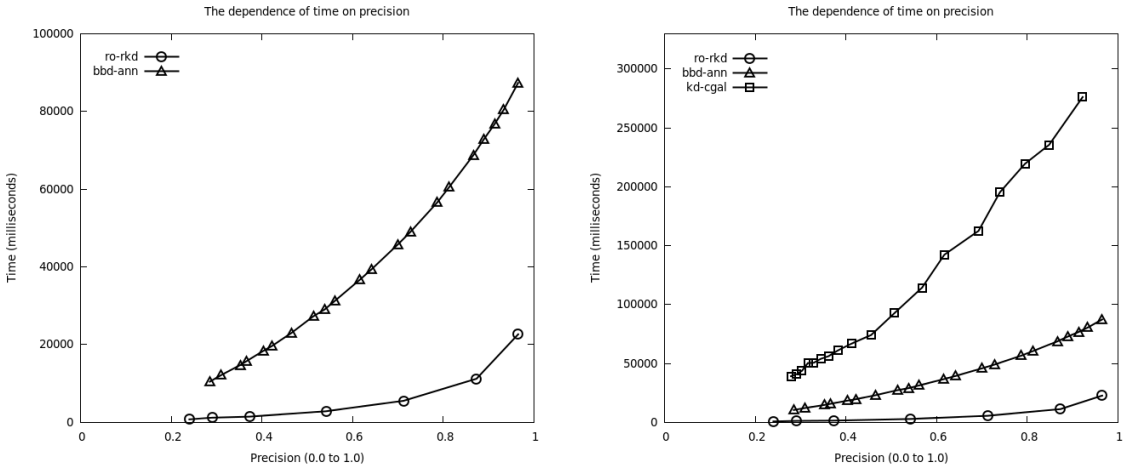
manifold describing the data. Even though the data set is significantly smaller in size $N_{GISETTE} < N_{SIFT} \Rightarrow 13e3 < 30e4$, the running times exhibited for similar precision rates are much higher. For values of approximately 0.7 the query times of Randomly-oriented RKD-trees for SIFT data are 40-50 milliseconds, whereas for the GISETTE data are 7e3-8e3 milliseconds. Two factors are important in the large margin between the run times, the dimension of the data and the structure of the data.

Data Name:	Gisette		
Data Set Characteristics:	Multivariate	**Number of Instances:**	13500
Attribute Characteristics:	Integer	**Number of Attributes:**	5000

Table 6.3: Gisette dataset

Once more Randomly-oriented RKD-trees outperform by an order of magnitude both the BBD-trees and k-d tree (CGAL) structures. As the precision increases the the running time margin increases by a significant about. The slopes of both outperformed structures are much larger than the one displayed by the ANN search algorithm built upon Randomly-oriented RKD-trees.

Throughout the different experiments the approximate nearest neighbor algorithm using Randomly-oriented RKD-trees consistently outperforms all other tested algorithms. The higher the precision we expect to obtain as a result the larger the expected gain in running time. Randomly-oriented RKD-trees offer a favorable alternative to the CGAL k-d tree nearest neighbor algorithm for medium and larger dimensional data sets. This is true for data sets with an underlying structure (ex. SIFT, Gisette) and without an underlying structure (ex. uniformly randomly generated data).



(a) (b)

Figure 6-4: (a) The graph shows that for the 5000-dimension GISETTE data set, the average searching time for 100 different queries when searching for the 20 nearest neighbors, Randomly-oriented RKD-trees outperforms the BBD-tree structure for all values of precision. (b) The graph shows that the order of magnitude difference between Randomly-oriented RKD-trees and the BBD-tree in average search-time is multiplied with respect to the search-time of the k-d tree (CGAL), especially for values of high precision.

Chapter 7

Conclusion

In this paper we introduce Randomly-oriented RKD-trees. We provide a bound (Theorem 7) of the number of levels required to decrease the size of the tree cells by a factor equal to $s \geq 2$. Theorem 7 may be used to improve the bound Theorem 2 of [32]. We further provide a bound on the smallest cell, in a cell created in Randomly-oriented RKD-trees, that completely contains a fixed ball B of radius R, by showing that the aspect ratio bound is $\mathrm{O}\big(d\sqrt{d}\log d\big)$.

We provide an implementation of the Randomly-oriented RKD-trees approximate nearest neighbor search (ANNS) algorithm. The implementation extends the *Computational Geometry Algorithms Library* (CGAL) [1] library. The implementation is provided with doxygen[1] documentation. Results show that search times are highly competitive against the current state of the art ANNS algorithms.

The bounds are the same as the bounds provided for RPTree-Max in [12], though they do not paint the entire picture regarding Randomly-oriented RKD-trees bounds. Attention needs to placed on the effect of the multi-tree greedy approach that make both RKD-trees and Randomly-oriented RKD-trees so effective in practice. Results show that Randomly-oriented RKD-trees implementations exhibit higher performance than BBD-trees. It is likely that the greedy selection of space partition branches offers an advancement that has not been fully analyzed in this paper.

Acknowledging that the open questions have first been raised for RPTree-Max

[1] `http://www.doxygen.org/`

[12], an open question may be left providing a stronger guarantee, bounding the query time based on the doubling dimension. The following may be shown to be true if the theoretical results of the multi-tree method is further pursued thus providing guarantees similar to the ones found for BBD-trees:

- **Bounded Depth**: depth of the tree should be $(\log n)^{\mathrm{O}(1)}$
- **Packing Guarantee**: $\left(\frac{R}{r}\right)^{(d \log \frac{R}{r})^{\mathrm{O}(1)}}$
- **Space Partitioning Guarantee**: size reduction by a factor s in $(d \log s)^{\mathrm{O}(1)}$ levels

Appendix A

Proofs

A.1 Proofs of Lemmas

A.1.1 Lemma 10

Proof of Lemma 10 taken from [32]. We begin by reporting an important assumption for the purposes of this proof. Let there exist a projection orthogonal to the arbitrary subspace V, and a subsequent projection to a random vector in subspace $V^{\perp}$, call it W. If we assume the size of W is large enough then any projection to this subspace is similar (and sufficient for our purposes) to projecting to a random vector in full space – the method RP-trees are based on.

Pick a minimum cover of $S \subset B(x, \Delta)$. From the definitions of doubling dimension the cover has at most 2^d balls of radius $\Delta/2$. Without loss of generality, we may assume that the centers of these balls lie in $B(x, \Delta)$. Each ball B induces a subset $S \cap B$; cover each such subset by 2^d smaller balls of radius $\Delta/4$, with centers in B. Continuing this process, the final result is a hierarchy of covers, at increasingly finer granularities.

Fix a ball $B(x_i, \Delta/2^k)$ at level k, and consider one of the balls, $B(x_j, \Delta/2^{k+1})$, that covers it. Let $W = V^{\perp}$. For the centers x_i and x_j it is true that $\|x_i - x_j\| \leq \Delta/2^k$.

From Lemma 8 and with $\beta = \sqrt{2(d + \log(2/\delta))}$ we have:

$$\begin{aligned}
&\mathbb{P}\left[\|\pi_v(x_i - x_j)\| \geq \beta\sqrt{k+1}\frac{\Delta/2^k}{\sqrt{D-d}}\right] \\
&\leq \mathbb{P}\left[\|\pi_v(\pi_W(x_i - x_j))\| \geq \beta\sqrt{k+1}\frac{\|\pi_W(x_i - x_j)\|}{\sqrt{D-d}}\right] \\
&\leq \frac{2}{\beta}\cdot\frac{1}{\sqrt{k+1}}\cdot \mathrm{e}^{-\left(\frac{\sqrt{2(d+\log(2/\delta))}^2}{2}(k+1)\right)} \\
&= \frac{2}{\beta}\cdot\frac{1}{\sqrt{k+1}}\cdot \mathrm{e}^{-((d+\log(2/\delta))(k+1))} \\
&\leq \frac{2}{\beta}\cdot\left(\frac{2}{3}\right)^k \cdot \mathrm{e}^{-(d(k+1))}\mathrm{e}^{-\log(2/\delta)^{(k+1)}} \\
&\leq \frac{2}{\beta}\cdot\frac{1}{\sqrt{k+1}}\cdot\left(\frac{\delta}{2}\right)^k\cdot\left(\frac{\delta}{2}\right)\cdot \mathrm{e}^{-(d(k+1))} \\
&\leq \frac{\delta}{\beta}\cdot\left(\frac{\delta}{2}\right)^k\cdot \mathrm{e}^{-(d(k+1))}
\end{aligned}$$

Take a union bound over all edges (x_i, x_j) in the tree. There are $2^{(k+1)d}$ edges between levels k and $k+1$, via use of the chain rule:

$$\begin{aligned}
&\mathbb{P}\left[\exists k:\ \exists x_i \text{ in level k with child } x_j : \|\pi_v(x_i - x_j)\| \geq \beta\sqrt{k+1}\frac{\Delta/2^k}{\sqrt{D-d}}\right] \\
&\leq \sum_{k=0}^{\infty} 2^{d(k+1)}\frac{\delta}{\beta}\left(\frac{\delta}{2}\right)^k \mathrm{e}^{-d(k+1)} \\
&\leq \frac{\delta}{\beta}\frac{1}{1-(\delta/2)} \leq \delta
\end{aligned}$$

Thus with probability at least $1-\delta$, for all k, every point $y \in S$ satisfies

$$\|\pi_v(y) - \pi_v(x)\| \leq \frac{\beta\Delta}{\sqrt{D-d}}\sum_{k=0}^{\infty}\frac{\sqrt{k+1}}{2^k} \leq \frac{3\Delta}{\sqrt{D-d}}\sqrt{2(d+\log(2/\delta))}.$$

□

A.1.2 Lemma 11

Proof. Fix a random point x chosen on some projected vector v. By lemma 8 the expectation that $\pi_v(x)$ is greater than the radius $O(\Delta/\sqrt{D})$ is:

$$\mathbb{E}\left[\mathbf{1}\left(|\pi_v(z)-\pi_v(x)|\geq c\cdot\tfrac{\|z-x\|}{\sqrt{D}}\right)\right]\leq \tfrac{2}{c}\mathrm{e}^{-c^2/2}\leq \delta\epsilon.$$

All this while seeting $c=\sqrt{2\log\frac{1}{\delta\epsilon}}$. Abriviate $F_x=\mathbf{1}\left(|\pi_v(z)-\pi_v(x)|\geq c\cdot\frac{\|z-x\|}{\sqrt{D}}\right)$

$$\mathbb{P}_{\pi_v(S)}\left[\mathbb{E}_\mu\left[F_x\right]\right]\leq\tfrac{\mathbb{E}_{\pi_v(S)}[\mathbb{E}_\mu[F_x]]}{\epsilon}=\tfrac{\mathbb{E}_\mu\left[\mathbb{E}_{\pi_v(S)}[F_x]\right]}{\epsilon}<\delta$$

□

A.1.3 Lemma 12

Proof of Lemma 12. Suppose μ is the mean of a normal ditribution and M is its median. The density funtion is then given by

$$f(x)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{2}\cdot\left(\frac{x-\mu}{\sigma}\right)^2\right),\ \text{for}\ -\infty<x<\infty.$$

$$\int_{-\infty}^{\mathrm{M}}f(x)\,\mathrm{d}x=\frac{1}{2}\Rightarrow\frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{\mathrm{M}}\exp\left(-\left(x-\mu\right)^2/2\sigma^2\right)\,\mathrm{d}x=\frac{1}{2}$$
$$\Rightarrow\frac{1}{\sigma\sqrt{2\pi}}\left[\int_{-\infty}^{\mu}\exp\left(-\left(x-\mu\right)/2\sigma\right)\,\mathrm{d}x+\int_{\mu}^{\mathrm{M}}\exp\left(-\left(x-\mu\right)/2\sigma\right)\,\mathrm{d}x\right]=\frac{1}{2}$$

For standard normal variate $Z=\frac{x-\mu}{\sigma}$, $\int_{-\infty}^{\mu}(\cdot)=\frac{1}{2}$. Thus, $\frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{\mu}\exp\left(-\left(x-\mu\right)/2\sigma\right)\,\mathrm{d}x$

$$\Rightarrow\frac{1}{2}+\frac{1}{\sigma\sqrt{2\pi}}\int_{\mu}^{\mathrm{M}}\exp\left(-\left(x-\mu\right)/2\sigma\right)\,\mathrm{d}x=\frac{1}{2}$$
$$\Rightarrow\frac{1}{\sigma\sqrt{2\pi}}\int_{\mu}^{\mathrm{M}}\exp\left(-\left(x-\mu\right)/2\sigma\right)\,\mathrm{d}x=0\Rightarrow\boxed{\mu=M}.$$

□

A.1.4 Lemma 14

Proof of Lemma 14. Following the same proof guidelines of Lemma 7 in [32], and using the result of the triangle inequality it is shown that $\|\pi_v(z-x)\| \leq 3\Delta/\sqrt{D}$ and $\|\mu_{\pi_v(S)} - \pi_v(x)\| \leq 3\Delta/\sqrt{D}$. Let $\beta = \sqrt{8}$ then from Lemma 9 *(b)*

$$\mathbb{P}(\|\pi_v(z-x)\| \geq \beta \frac{\|z-x\|}{\sqrt{D}}) \leq \frac{2}{\beta} \mathrm{e}^{-\beta^2/2} \leq \frac{1}{\sqrt{2}\mathrm{e}^4}$$

Given that $\not{d} < \frac{D}{9}$ *we get*

$$\beta \frac{\|z-x\|}{\sqrt{D-\not{d}}} \leq \sqrt{8} \frac{\|z-x\|}{\sqrt{D-\frac{D}{9}}} = 3\frac{\|z-x\|}{\sqrt{D}}$$

showing that

$$\mathbb{P}(\|z-x\| \geq 3\frac{\|z-x\|}{\sqrt{D}}) \leq \frac{1}{-}$$

Continuing and setting $\delta = 2/\mathrm{e}^{9/2}$

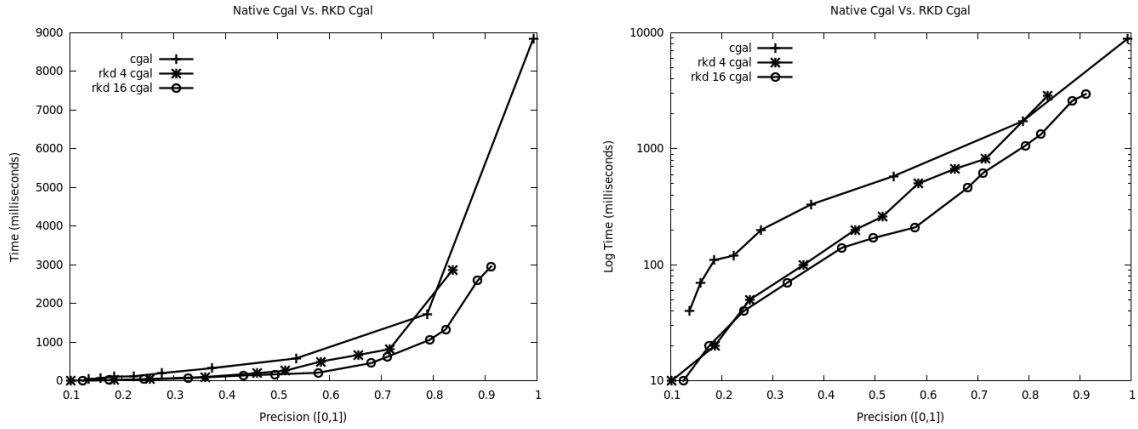
$$\begin{aligned} \|\mu_{\pi_v(S)} - \pi_v(x)\| &\leq \frac{\Delta}{\sqrt{D}} \sqrt{2log(\frac{2}{\delta})} \\ &\leq \frac{3\Delta}{\sqrt{D}} \end{aligned}$$

The total failure probability is upper bounded by: $\frac{1}{\sqrt{2}\mathrm{e}^4} + \frac{2}{\mathrm{e}^{9/2}} < \frac{1}{20}$

□

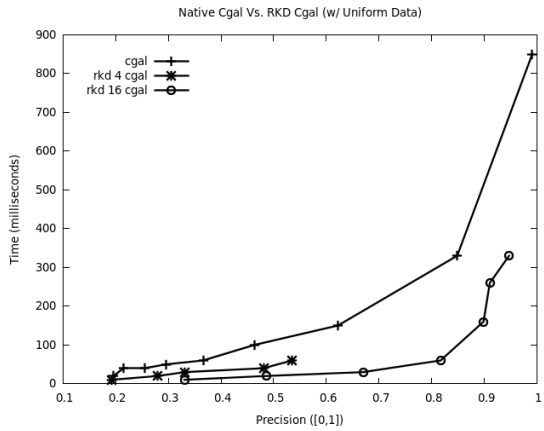
Appendix B

Supplementary Experimental Results

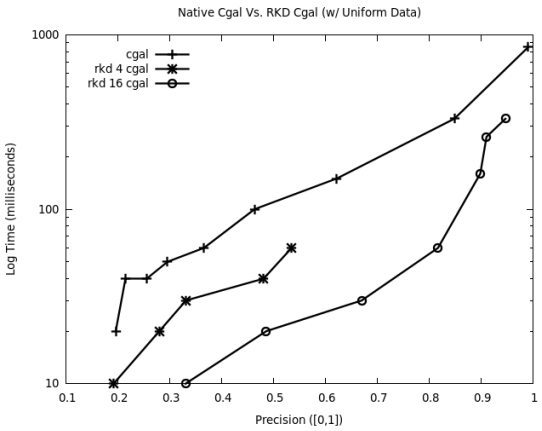


(a) Time (milliseconds) vs. Precision [0,1] (b) Lg(Time) vs. Precision [0,1]

Figure B-1: (a) Compare the search time of 4 and 16 tree structures of CGAL Randonly-oriented RKD-trees with the native CGAL ANN search, of randomly generated Poisson distributed data. (b) The log scaled y-axis of the comparison illustrated in figure (a).

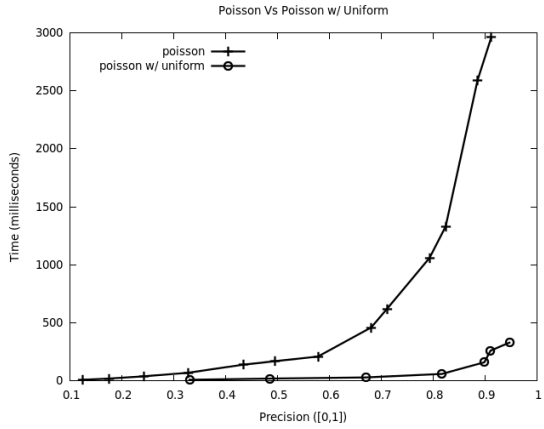


(a) Time (milliseconds) vs. Precision [0,1]



(b) Lg(Time) vs. Precision [0,1]

Figure B-2: (a) Compare the search time of 4 and 16 tree structures of CGAL Randonly-oriented RKD-trees with the native CGAL ANN search, of randomly generated Poisson in addition to high variance and uniformly distributed data. (b) The log scaled y-axis of the comparison illustrated in figure (a).



(a) Time (milliseconds) vs. Precision [0,1]

Figure B-3: (a) Compare 16 tree Randonly-oriented RKD-trees search result times for Poisson and Poisson w/ uniform distributed datasets.

Appendix C

User Manual

C.1 Introduction

The spatial searching package implements exact and approximate query searching by providing implementations of algorithms supporting

- Nearest and furthest neighbor searching
- Exact and approximate searching
- Approximate k-nearest and k-furthest searching
- Query items representing points (extending to spatial objects)

A set of N data points in D-dimensional space is given. The points in Euclidean space of dimension D are identified with tuples of D real numbers, with the Cartesian product $\mathbb{R}^D$. They are preprocessed into a space partitioning data structure, such that, given any query item q the set of points can be browsed efficiently. The *Randomly-Oriented RKD-trees* spatial searching package is designed for data sets that are represented in medium to large dimension D with the objective of performing highly time-efficient and accurate approximate spatial queries in pursuit of one or more approximate nearest nearest neighbors. The space is intended to be small enough to store the search structure in main memory.

C.1.1 Build Randomly-oriented random k-d trees

`Random_kdtree_k_neighbor_search` defines the functionality for performing approximate nearest search. The main input is the vector of `RKd_tree` structures of `Point` and index tuples. For each tree the root function is called. The function builds the tree and returns the root node to the `RKd_tree`. For each tree the point set is shuffled and a new random tree is built. The leaf nodes reference a single node. Each node is split in accordance to the splitting definition passed as a template parameter.

Random rotation

The class Prior to defining the package associated with the build process, we present the random rotation method. It is responsible for transforming the data so that RKD-trees are assured to adapt to the intrinsic dimensionality of the data.

C.1.2 Neighbor Searching

For each query point a search is performed. The point traverses each tree down to a leaf node containing a single point. The point is passed into a sorted queue if the distance is less than the distance of the longest distance in the queue. As the queue traverses the tree a record of the branches not taken is kept in a heap if the branch possibly contains points nearer points than the ones in the queue. A search is performed for all trees and the best matches are kept in a single sorted queue which is returned as the output. The size of the queue is equal to the number of nearest neighbors the query is performed for. Once a first traversal is performed on each tree the heap is 'popped' one by one and the branch is traversed down to a leaf node referencing a point. This process is performed until the maximum number of searches is performed or until the error factor specifies that no more searches need to be performed and the resulting queue can be returned as output.

C.2 Splitting Rules

In addition to the splitting rules of dD Spatial Searching, a user may, select one of the following splitting rules, which are tailord for use in a RKD-tree:

`mean_split`

This splitting rule cuts a rectangle through an approximation of its mean orthogonal to one of its high variance sides.

`median_split`

This splitting rule cuts a rectangle through a random point within a user defined range centered around its median orthogonal to one of its high variance sides.

C.3 Example Programs

Listing C.1: code/example1.c

```
#include <cstdint>
#include <vector>
#include <algorithm>
#include <chrono>
#include <cmath>
#include <time.h>
#include <utility>

#include <CGAL/Cartesian_d.h>
#include <CGAL/point_generators_d.h>
#include <CGAL/Kd_tree.h>
#include <CGAL/Search_traits_d.h>
#include <CGAL/Orthogonal_k_neighbor_search.h>
#include <CGAL/Euclidean_distance.h>
#include <CGAL/property_map.h>
#include <CGAL/basic.h>
#include <CGAL/Search_traits_d.h>
#include <CGAL/Search_traits_adapter.h>

#include <boost/iterator/zip_iterator.hpp>

#include "Random_kdtree_k_neighbor_search.h"

typedef CGAL::Cartesian_d<double> K;
typedef K::Point_d Point_d;
typedef CGAL::Search_traits_d<K> Traits;

typedef CGAL::Random_points_in_cube_d<Point_d> Random_points_iterator;
typedef CGAL::Counting_iterator<Random_points_iterator> N_Random_points_iterator;
typedef CGAL::Plane_separator<K> SpatialSeparator;
```

```
typedef CGAL::Midpoint_of_max_spread<Traits, SpatialSeparator> Midpoint_Seperator;
typedef CGAL::Kd_tree_node<Traits, Midpoint_Seperator, bool> Tree_Node;
typedef CGAL::Euclidean_distance<Traits> Distance;

// RKd_tree_neighbor_search tests
typedef boost::tuple<Point_d,int> Point_and_int;
typedef CGAL::Search_traits_d<K> Traits_base;
typedef CGAL::Search_traits_adapter<Point_and_int,
  CGAL::Nth_of_tuple_property_map<0, Point_and_int>,
  Traits_base> Traits_t;

typedef CGAL::Random_kdtree_k_neighbor_search<Traits_t> K_neighbor_search_t;
typedef K_neighbor_search_t::Tree Tree_t;
typedef K_neighbor_search_t::Distance Distance_t;

#define EXIT 0

using namespace std;

int main(int argc, char **argv) {
   const unsigned int K = 1;
   const unsigned int dim = 3;
   const unsigned int n_points = 10;
   double size = 100.0;
   CGAL::Random_points_in_cube_d<Point_d> gen (dim, size);

   std::vector<Point_d> points;
   std::vector<int> indices;

   // generate n_points number of points along
   // with its associated index
   for(unsigned int i=0; i<n_points; ++i) {
      points.push_back((*gen++));
      indices.push_back(i);
   }

   const unsigned int n_trees = 2;
   std::vector<Tree_t> trees;
   trees.resize(n_trees);

   // build n_trees number of trees
   for(unsigned int i=0; i<n_trees; ++i) {
      Tree_t* tree =
         new Tree_t(boost::make_zip_iterator(boost::make_tuple( points.begin(), indices.begin() )),
         boost::make_zip_iterator(boost::make_tuple( points.end(), indices.end() )) );
      trees[i] = *tree;
   }

  Point_d query((*gen++)); // random query point

  K_neighbor_search_t search( trees, query, K);

  // nearest neighbor indices
  std::vector<std::vector<int>> result;
  result.push_back(search.get_resulting_points());

  cout << "The point with indeces: ";
  for(auto res_it : result) {
   for(auto neigh_it : res_it)
      std::cout << neigh_it << " ";
```

```
  }
  std::cout << std::endl;

   cout << "each at a distance ";
   // nearest neighbor distances
   for(auto dist_it : search.get_resulting_distances()) {
      std::cout << dist_it << " ";
   }
   cout << "from the query point";

  return EXIT;
}
```

Bibliography

[1] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[2] Sunil Arya, Theocharis Malamatos, and David M. Mount. Space-efficient approximate voronoi diagrams. In *STOC*, pages 721–730, 2002.

[3] Sunil Arya and David M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas.*, pages 271–280, 1993.

[4] 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.

[5] Patrice Assouad. Plongements lipschitziens dans $\mathbb{R}^n$. (Lipschitz embeddings into $\mathbb{R}^n$). *Bull. Soc. Math. Fr.*, 111:429–448, 1983.

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

[7] K.L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 226–232, Nov 1983.

[8] R. Scott Cost and Steven Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–78, 1993.

[9] Chang da Bei and R.M. Gray. An improvement of the minimum distortion encoding algorithm for vector quantization. *Communications, IEEE Transactions on*, 33(10):1132–1133, Oct 1985.

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

[11] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004.

[12] Aman Dhesi and Purushottam Kar. Random projection trees revisited. *CoRR*, abs/1010.3812, 2010.

[13] Persi Diaconis and Mehrdad Shahshahani. The subgroup algorithm for generating uniform random variables. *Probability in the Engineering and Informational Sciences*, 1:15–32, 1 1987.

[14] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1987.

[15] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[16] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[17] Jerome H. Friedman, Forest Baskett, and Leonard J. Shustek. An algorithm for finding nearest neighbors. *IEEE Trans. Computers*, 24(10):1000–1006, 1975.

[18] Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[19] Isabelle Guyon, Steve Gunn, Asa Ben-Hur, and Gideon Dror. Result analysis of the nips 2003 feature selection challenge. In L.K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 545–552. MIT Press, 2005.

[20] Piotr Indyk. Nearest neighbors in high-dimensional spaces. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press LLC, Boca Raton, FL, 2nd edition, April 2004.

[21] W. B. Johnson and J. Lindenstrauss. Extensions of Lipschitz maps into a Hilbert space. 1984.

[22] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[23] Vincent Lepetit and Pascal Fua. Keypoint recognition using randomized trees. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(9):1465–1479, 2006.

[24] David G. Lowe. Object recognition from local scale-invariant features. In *ICCV*, pages 1150–1157, 1999.

[25] Marvin Minsky and Seymour Papert. Perceptrons: An introduction to computational geometry. 1969.

[26] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP (1)*, pages 331–340, 2009.

[27] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 71–79, 1995.

[28] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13-15, 1975*, pages 151–162. IEEE Computer Society, 1975.

[29] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*, 2008.

[30] Robert F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.

[31] Hans Tangelder and Andreas Fabri. dD spatial searching. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.4 edition, 2014.

[32] Santosh Vempala. Randomly-oriented k-d trees adapt to intrinsic dimension. In Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan, editors, *FSTTCS*, volume 18 of *LIPIcs*, pages 48–57. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[33] Charles M Werneth, Mallika Dhar, Khin Maung Maung, Christopher Sirola, and John W Norbury. Numerical gramschmidt orthonormalization. *European Journal of Physics*, 31(3):693, 2010.