Nearest Neighbor Search in Multidimensional Spaces

Depth Oral Report

Panayiotis Tsaparas

Department of Computer Science

University of Toronto

tsap@cs.toronto.edu

June 10, 1999

Abstract

The Nearest Neighbor Search problem is defined as follows: given a set P of n points, preprocess the points so as to efficiently answer queries that require finding the closest point in P to a query point q. If we are willing to settle for a point that is almost as close as the nearest neighbor, then we can relax the problem to the approximate Nearest Neighbor Search. Nearest Neighbor Search (exact or approximate) is an integral component in a wide range of applications that include multimedia databases, computational biology, data mining, and information retrieval. The common thread in all these applications is similarity search: given a database of objects, we want to return the object in the database that is most similar to a query object. The objects are mapped onto points in a high dimensional metric space , and similarity search reduces to a nearest neighbor search. The dimension of the underlying space may be in the order of a few hundreds, or thousands; therefore, we require algorithms that perform efficiently even for spaces of high dimension.

Due to its importance, the Nearest Neighbor Search problem has been a subject of research for many decades, and in many different fields. In this work we survey some of the past and recent results that marked the evolution of the problem. The report starts with the optimal solutions for low-dimensional spaces, then it goes through the solutions for the exact and approximate problem for spaces of arbitrary dimension, and then it finishes off with the recent attempts to prove lower bounds in order to specify the hardness of the problem. We also investigate the problem from the practitioner's point of view, and we provide an overview of the indexing problem for Nearest Neighbor Queries. Finally, we identify some open questions, and interesting problems.

Contents

1 Introduction **1**

2 Problem Definition **2**

3 Tools from Computational Geometry **4**

3.1 Hyperplanes and half-spaces 4

3.2 Convex Polytopes 4

3.3 Arrangements of hyperplanes 5

3.4 Voronoi diagrams 5

4 Nearest Neighbor Search in low dimensions **7**

5 Exact Nearest Neighbor Search in d-dimensional Spaces **10**

5.1 Algorithms based on Voronoi diagrams 10

5.1.1 The First Attempt 10

5.1.2 Random Sampling Algorithms 11

5.2 Space Partition Algorithms 14

5.3 Summary of the results 15

6 Approximate Nearest Neighbor Search in d-dimensional Spaces **16**

6.1 The Early Attempts 16

6.2 Recent Algorithms 18

6.2.1 Randomized Testing 18

6.2.2 The Kleinberg algorithms 18

6.2.3 The Hamming Space 20

6.2.4 Approximate Nearest Neighbor Search in the Euclidean space 22

6.3 Summary of the results 23

7 A practical approach to Nearest Neighbor Search **23**

7.1 Algorithms for Nearest Neighbor Search 24

7.1.1 The HS algorithm 24

7.1.2 The RKV algorithm 25

7.2 Indexes for Nearest Neighbor queries 26

7.3 Evaluation of indexes 29

8 Lower Bounds for the Nearest Neighbor Search Problem **31**

8.1 The cell probe model 31

8.2 The semi-group model 33

8.3 The bucketing model 35

8.4 The indexing model 36

9 Conclusions and Discussion **38**

A Range Spaces and VC dimension **47**

B Parametric Search **49**

C The Kushilevitz et al. algorithm **49**

1 Introduction

The problem of *Nearest Neighbor Search* (NNS) can be defined informally as follows: given a database of n points in some metric space, preprocess the points so as to efficiently answer queries for finding the point in the database closest to a query point. This problem, also known as the *post office problem*, has been of great theoretical interest in the area of Computational Geometry, and it has been a research topic for many decades. Recently, it has drawn more attention due to its applications in a wide range of applications that involve the notion of *similarity search*, that is, given a set of objects, return the object most similar to a query object. Examples of such areas include information retrieval [76, 74, 32, 19], multimedia applications [39, 41, 73], DNA matching [4, 72, 63, 84], data compression [49], pattern recognition [36, 31], and machine learning [30].

Typically, the objects are mapped into a multidimensional space of *features*, and similarity search is reduced to a Nearest Neighbor Search in this feature space. In most applications, the resulting feature space is of very high dimension. For example in information retrieval, documents and queries are represented as vectors in the space of terms, a space of thousands of dimensions. Even though there exist techniques such as LSI [32, 19, 13, 38], Principal Component Analysis [54, 38], or Karhunen-Loeve transform [38], that reduce the dimensionality of the space, the resulting dimension is still in the order of a few hundreds.

The situation is similar in the case of image retrieval systems such as IBM's QBIC [39, 41] and MIT's Photobook [73]. Images are mapped to a set of features containing color, shape, texture and object relation information. The dimensionality of each feature is typically large, so the resulting space has dimension of a few hundreds. This is expected to increase as more and more sophisticated techniques for feature extraction are becoming available.

Similar examples can be given in the area of data warehousing and data mining. It seems likely that in the future the dimensionality of the search space will increase, as systems and data sets become more complex. Thus, the need for an efficient solution for the NNS problem in high dimensions becomes imperative.

The problem has been solved optimally, or near optimally, for the case of low dimensions, but for large dimensions there has been very little progress. Specifically, most solutions either create data structures that require storage space exponential in the dimension, or require query time that is not much better than a linear scan of the data points. In fact for dimension $d > \log n$, a brute force search is usually the best choice both in theory [27, 6], and in practice [14, 86], a predicament commonly referred to as the *curse of dimensionality* [1].

In an attempt to circumvent the curse of dimensionality, researchers resorted to *approximate* nearest

[1]The term "curse of dimensionality" is a broadly used term that describes the case when the performance of a system depends exponentially on the dimensionality of the underlying space.

neighbor search. Since the selection of features as well as the distance metric in most applications is of a heuristic nature, an approximate search is as good as the exact search for all practical purposes. For some time it seemed that the the problem of approximate search might be as hard as the exact search. Recently, there has been some progress towards removing the curse of dimensionality. The solutions produced however, are thus far purely of theoretical interest.

The NNS problem is of independent interest to the database community. In this context we are interested in constructing an index for the blocks in secondary memory that store the database points that will enable us to find the nearest neighbor of a point by accessing a small number of blocks. Current indexes do not cope well with spaces of high dimension, and it can be shown that for commonly used indexes in the worst case, and for typical data distributions, all blocks of the index have to be accessed in order to determine the nearest neighbor. Recently, there has been an increasing interest in creating indexes directed specifically to the NNS problem, as well as in providing some theoretical guarantees that these indexes will perform well. Note that in this context the cost of the search is no longer measured in terms of operations, but in terms of blocks retrieved from the secondary memory.

In this report we survey the NNS problem, for both the exact and the approximate case. In section 2 we formally define the problem. In section 3 we introduce some definitions and techniques from computational geometry, which underlie many of the existing solutions for the problem. In section 4, we consider the problem in low dimensions, and we present two optimal solutions for the 2-dimensional case. In section 5 we investigate the exact NNS problem, while in section 6 we look into the approximate NNS problem. In section 7 we view the problem from a more practical viewpoint, and we investigate indexing for Nearest Neighbor queries. In section 8 we present some models for lower bound analysis of the problem. Finally, the last section concludes the report.

2 Problem Definition

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n points (also referred to as *sites*) in a metric space $\mathcal{M} = (\mathcal{V}, d(\cdot,\cdot))$, where $\mathcal{V}$ is a d-dimensional vector space, and $d : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$ denotes a (dis)similarity measure. We give the following definitions.

Definition 1 *Given a set P of n points in $\mathcal{M}$, and a query point $q \in \mathcal{V}$, a nearest neighbor of q is a point $p \in P$, such that for any other point $p' \in P$,*

$$d(p,q) \leq d(p',q).$$

Definition 2 *Given a set P of n points in $\mathcal{M}$, a parameter $\varepsilon > 0$, and a query point $q \in \mathcal{V}$, an $(1+\varepsilon)$-approximate nearest neighbor of q is a point $p \in P$,such that for any other point $p' \in P$,*

$$d(p,q) \leq ((1+\varepsilon)d(p',q).$$

The parameter ε is called the approximation factor.

Nearest Neighbor Search is the problem of returning the nearest neighbor of the query point q. If we allow insertions and deletions of points, we have the *dynamic* NNS problem. If we consider the database of points to be fixed then we have the *static* NNS problem. In this report we consider only the static NNS problem. The static NNS problem (exact or approximate) amounts to devising a data structure for the points in P, so that given a query point q we can compute the nearest neighbor efficiently. The efficiency of a solution is evaluated with respect to two measures: the amount of space required to store the data structure, and the cost of retrieving the nearest neighbor of the query point. The cost of the query depends on the cost model that we employ. In the main memory model, we assume that all data resides in main memory, and the cost of the query is the number of CPU operations performed by the search algorithm. In the external memory model we assume that the data has to be brought from the hard disk to the main memory, and the cost of a query is the number of disk accesses performed by the search algorithm. Ideally, an optimal algorithm should use $\mathcal{O}(dn)$ space, and perform $\mathcal{O}(\log n)$ operations. A satisfactory solution would be one that achieves polynomial space on n and d, and has query time that is polynomial on d and $\log n$. For practical purposes the requirements are sometimes even stricter: a viable solution should have space linear on d and n, for a small multiplicative constant.

A variety of different NNS problems can be defined depending on the choice of the vector space $\mathcal{V}$, and the distance metric d. The most extensively studied space is the real vector space $\mathbb{R}^d$. The most commonly used distance measure for $\mathbb{R}^d$ is the Euclidean distance L_2. However, other Minkowsky metrics L_p, such as the Manhattan distance L_1, or the maximum norm L_∞, are also widely used [55, 10, 48].

Another interesting case is the space Σ^d, the space of all strings of size d over some alphabet Σ. The distance between two strings is usually defined as the number of coordinates in which the two strings differ. In some biological applications the distance between two stings is defined as the *edit distance*. Given a set of operations, the edit distance is defined as the cost of the least expensive sequence of operations that converts one string to the other.

A popular space is the metric space (H^d, h): the Hamming cube of dimension d, equipped with the Hamming distance. The Hamming distance between two binary strings is defined as the number of coordinates into which they differ. The Hamming space can be thought of as a special case of the real vector space under the L_1 norm, or the space of strings as it is defined above. This is an appealing case because of its simplicity, and the wide range of applications it has.

There are other interesting variations of the NNS problem. Probably the most straightforward generalization of the problem is the k-Nearest Neighbor Search, where we return the k closest points to the query point. We can define approximate versions of this problem. We can also define a restricted NNS(λ) problem, where we require a nearest neighbor of a query point, if that nearest neighbor lies within distance λ. This can be extended to the λ-*neighborhood* problem, where we return all nearest neighbors that lie within radius λ from the query point. Naturally, we are also interested in the decision version of the problem, which we denote as DNNS(r): return YES if the nearest neighbor lies within distance r, and NO otherwise. There are also other slightly different formulations of the decision problem. The decision problem is interesting with respect to proving lower bounds for the NNS problem, and it can also be used as a subroutine for finding the (exact or approximate) nearest neighbor, using techniques that we will describe later.

3 Tools from Computational Geometry

In this section we introduce some definitions and tools from the area of computational geometry that we will be using throughout the report. For the remainder of this section we consider the vector space to be $\mathbb{R}^d$, and the distance metric to be the Euclidean norm L_2. The presentation in this section follows closely the presentation in [70].

3.1 Hyperplanes and half-spaces

We define a *hyperplane* h in $\mathbb{R}^d$ as the set of points in $\mathbb{R}^d$ that satisfy a linear equality of the form $a_1x_1 + a_2x_2 + \ldots + a_dx_d = a_0$, where x_j's denote the coordinates in $\mathbb{R}^d$, the coefficients a_j's are arbitrary real numbers, and $a_1, \ldots, a_d$ are not all zero. A hyperplane in a space of dimension d has dimension $d - 1$. For $d = 2$, a hyperplane is simply a line, and for $d = 3$, a hyperplane is a plane.

A hyperplane h partitions the remaining space into two areas; the set of points such that $a_1x_1 + a_2x_2 + \ldots + a_dx_d > a_0$ and the set of points such that $a_1x_1 + a_2x_2 + \ldots + a_dx_d < a_0$. These areas are called *half-spaces*, and the hyperplane h is called the *bounding hyperplane* of these half-spaces.

3.2 Convex Polytopes

The intersection of a set of half spaces in $\mathbb{R}^d$ defines a d-dimensional *convex polytope*. A special case of convex polytopes arises when all half-spaces extend to infinity at some fixed direction. In this case the intersection of the half-spaces is called an *upper convex polytope*. If we assume by convention that every hyperplane uniquely defines a half-space, then we may consider an upper convex polytope to be defined by the set of the bounding hyperplanes.

We now define the *faces* of a d-dimensional convex polytope. The d-face of the polytope is the convex polytope itself. The boundary of the convex polytope is a collection of $(d-1)$-dimensional convex polytopes, which are called the $(d-1)$-faces, or *facets* of the convex polytope. Proceeding inductively we can define a j-face for all $0 \leq j \leq d$ as a j-dimensional convex polytope that appears on the boundary of a $(j+1)$-face. The 1-faces of the polytope are also called *edges* of the polytope, and the 0-faces are called *vertices* of the polytope.

A d-dimensional polytope on $d+1$ vertices is called a *simplex*. For example, a 2-dimensional simplex is a triangle. A simplex has the property that every j-face is a j-dimensional simplex.

3.3 Arrangements of hyperplanes

Let H be a set of hyperplanes in $\mathbb{R}^d$. The *arrangement* $A(H)$ is the decomposition of the space $\mathbb{R}^d$ into regions of varying dimensions. The hyperplanes in H induce a natural partition of the space into d-dimensional convex regions that are called *cells*, or *d-faces*. Each cell is a d-dimensional convex polytope. A j-face of any such cell is a j-face of $A(H)$. The set of faces of the arrangement $A(H)$ is the set of all faces of the cells of $A(H)$. We define the *complexity* of $A(H)$ to be the total number of faces of any dimension. The complexity of any arrangement of n d-dimensional hyperplanes is $\mathcal{O}^*(n^d)$, where the $\mathcal{O}^*(\cdot)$ notation denotes hidden factors that grow at least as fast as 2^d (in this case there is a hidden $d!$ factor).

An arrangement $A(H)$ can be represented by its *facial lattice*. This lattice contains a node for each j-face of $A(H)$. A node for a j-face f is connected to all nodes for the $(j-1)$-faces that appear on the boundary of the face f.

Given an arrangement of hyperplanes $A(H)$, and a query point q, the *point location* problem is defined as the problem of identifying the face of the arrangement that contains the query point q.

3.4 Voronoi diagrams

Let P be a set of sites on the real line $\mathbb{R}$. For any two consecutive sites find the middle-point between them. The interval between any two middle-points defines an area that contains all points that are closest to some site p. This notion is generalized in higher dimensions by introducing *Voronoi diagrams*. The *Voronoi region* of some site p, is defined as the area that contains all points in $\mathbb{R}^d$ that are closest to point p.

Formally, the Voronoi region $\mathcal{V}(p)$ of some site p is defined as the intersection of the half-spaces that contain p, which are bounded by the hyperplanes that are the perpendicular bisectors of the lines that connect p with all the other sites. The partition of $\mathbb{R}^d$ into the regions $\mathcal{V}(p)$ for all $p \in P$ is called the Voronoi diagram $\mathrm{Vor}(P)$ of P (see figure 1).

Each Voronoi region is a d-dimensional convex polytope, and it is called a *d-face*, or simply a *cell* of the Voronoi diagram. The j-faces of the Voronoi regions are the j-faces of the Voronoi diagram. The *complexity* of the Voronoi diagram is defined as the total number of faces of any dimension. The complexity of the Voronoi diagram in d dimensions is $\mathcal{O}^*(n^{\lceil d/2 \rceil})$. The time to construct the Voronoi diagram is $\mathcal{O}(n \log n + n^{\lceil d/2 \rceil})$. A Voronoi diagram is usually represented using a facial lattice, similar to the one described for the arrangements of hyperplanes.

Given the Voronoi diagram $\mathrm{Vor}(P)$ of a set of sites P, and a query point q, the NNS problem is

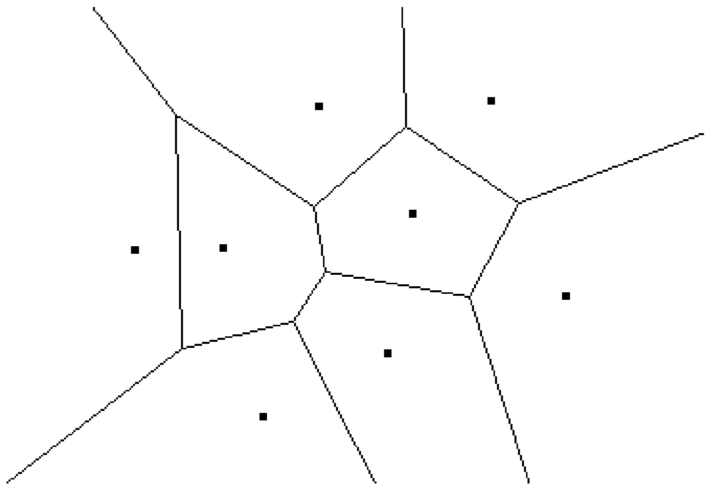

Figure 1: A Voronoi diagram

reduced to the problem of locating the Voronoi region that contains the query point q. Point location in a Voronoi diagram can also be viewed as a special case of point location in an arrangement of hyperplanes $A(H)$, where H is the set of perpendicular bisectors that define the Voronoi diagram. Given the face of $A(H)$ that contains q we can easily identify the corresponding face of the Voronoi diagram.

There is a strong connection between Voronoi diagrams in $\mathbb{R}^d$, and convex polytopes in $\mathbb{R}^{d+1}$. Every Voronoi diagram in $\mathbb{R}^d$ can be obtained by vertically projecting the boundary of an upper convex polytope in $\mathbb{R}^{d+1}$. Let P be the set of sites in $\mathbb{R}^d$. Identify the space $\mathbb{R}^d$ with the hyperplane $x_{d+1} = 0$ in $\mathbb{R}^{d+1}$. Take the vertical projection of every point p in P onto the point $\bar{p}$ on the unit parabola

$$U : x_{d+1} = x_1^2 + x_2^2 + \ldots + x_d^2.$$

Let $h(\bar{p})$ be the hyperplane that is tangent to the parabola U, and passes from $\bar{p}$. Let H be the set of hyperplanes $h(\bar{p})$, for all $p \in P$. Denote by $C(H)$ the upper convex polytope defined by the hyperplanes in H. For any site $p \in P$, the Voronoi region of p is obtained by vertically projecting the facet of $C(H)$ that lies on the hyperplane $h(\bar{p})$ onto the hyperplane $x_{d+1} = 0$. Thus, the Voronoi diagram $\mathrm{Vor}(P)$ is obtained by the vertical projection of the boundary of the convex polytope $C(H)$. (see figure 2)

Given this connection between Voronoi diagrams and convex polytopes, the point location problem can be reduced to the *ray shooting* problem. Given the upper convex polytope $C(H)$ in $\mathbb{R}^{d+1}$, and a ray emanating from infinity and directed to the query point q in the negative x_{d+1} direction, identify the first hyperplane of the the polytope $C(H)$ that is intersected by the ray. It is not hard to see that q belongs to the Voronoi region of the site that corresponds to the intersected hyperplane.

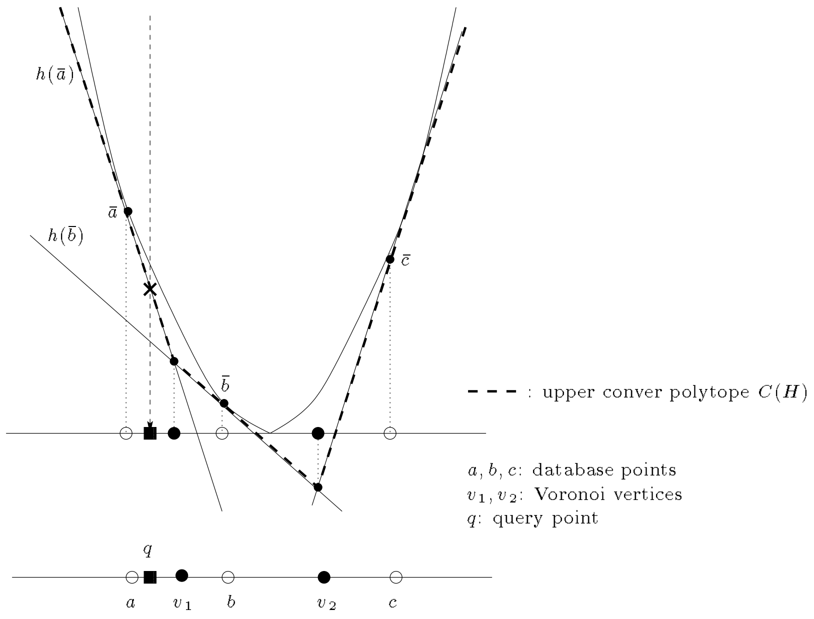

Figure 2: The relation between Voronoi diagrams and upper convex polytopes

4 Nearest Neighbor Search in low dimensions

The problem of Nearest Neighbor Search has been solved optimally for spaces of low dimension. For the 1-dimensional case the nearest neighbor of a query point can be easily found by performing a binary search. Take the points in P and sort them. For any two consecutive points find the middle-point between them. Given the query point q, perform a binary search on the middle-points. The result of the binary search will be an interval that determines the point closest to q. Thus, using $\mathcal{O}(n)$ space, and with $\mathcal{O}(n \log n)$ preprocessing time, we have a data structure with $\mathcal{O}(\log n)$ search time.

Dobkin and Lipton [33] proposed the following natural generalization of the binary search to two dimensions. Given the point set P, create the Voronoi diagram of P. Using the plane sweep algorithm [70] this takes $\mathcal{O}(n \log n)$ time. Then sort all the vertices of the Voronoi diagram according to their x coordinate. Any two consecutive x coordinates define a *slab* of the two dimensional space. This slab is intersected by a set of edges of the Voronoi diagram which do not intersect each other within the slab. Thus, we can define an ordering of these edges. Given now a query point q, we first perform a binary search on the x coordinates to locate the slab in which q belongs. Then we perform a binary search on the edges to locate the position of the point in the slab. The query time is the time for the two binary searches. Since in a two dimensional Voronoi diagram the number of vertices and the number of edges is $\mathcal{O}(n)$, the search time for any query is $\mathcal{O}(\log n)$. The necessary space however, is $\mathcal{O}(n^2)$, since there are $\mathcal{O}(n)$ slabs, each containing $\mathcal{O}(n)$ edges in the worst case.

The storage requirements can be reduced to $\mathcal{O}(n)$ by introducing the concept of *persistence* [80]. A dynamic data structure is called *persistent* if it allows accesses to any of its previous versions. The main idea is to exploit the fact that the location structures for adjacent slabs are similar – the edges with right endpoint on the slab boundary are removed and the edges with left endpoint are added. We "sweep" the plane by a line $\ell(x)$ vertical to the x axis. We regard the sweeping direction as the time axis, and the x coordinates of the vertices as points in "time". A balanced binary tree is used to store the edges in each slab, which is updated every time the line $\ell(x)$ encounters a vertex of the subdivision. We use a persistent data structure to keep the evolution of the tree over "time". By keeping duplicate copies of pointers, and copies of nodes that are updated, we can store a persistent data structure for the slabs in linear space, and keep the time in $\mathcal{O}(\log n)$. This technique is called *limited node copying persistence* [81].

A different approach to the NNS problem is to consider it as a special case of the problem of point location in *monotone planar subdivisions* [37]. A planar subdivision is a partition of the plane into vertices, edges, and polygonal faces. A monotone subdivision is a planar subdivision whose faces are x-monotone polygons, i.e. the intersection of any face with any vertical line is connected. Note that a Voronoi diagram defines a monotone planar subdivision.

Define now a *separator* (or separating chain) as a connected union of edges and vertices of the subdivision, such that it meets any vertical line in exactly one point. Let s be a separator, and let f_1 and f_2 be two faces of the subdivision. We write $f_1 \prec s$ if f_1 lies below the separator, and $s \prec f_2$ if f_2 lies above the separator. Note that this defines an ordering on the faces: $f_1 \prec f_2$; face f_1 is below face f_2 (or face f_2 is above face f_1). If ℓ is the total number of faces of the subdivision, then we define a *complete family of separators* to be a set of $\ell - 1$ separators, such that

$$f_1 \prec s_1 \prec f_2 \prec s_2 \prec \ldots \prec s_{\ell-1} \prec f_\ell.$$

Note that this imposes a total ordering on the faces of the subdivision. It is proven [37] that every monotone subdivision admits a complete family of separators.

Given a complete family of separators a data structure for locating a query point q in the monotone subdivision can be constructed. Store the faces of the subdivision as leaves of a binary search tree. The internal nodes of the tree will be the separators that separate the faces (see figure 3). Given a node of the tree, and the corresponding separator s_i, perform a binary search on the edges of the separator to locate the edge cut by the vertical line that passes through q. Then check if q lies above or below the edge. If it lies below the edge move to the left child of the node, otherwise move to the right child. Since the depth of the tree is $\log n$ and we perform a binary search at each node, the query time of the data structure is $\mathcal{O}(\log^2 n)$. Note that once we have checked the query point against some edge e, we never reconsider this edge again. So, we store each edge exactly once, at the first separator node containing it that we encounter as we traverse down the tree. Thus the data structure uses $\mathcal{O}(n)$ storage space.

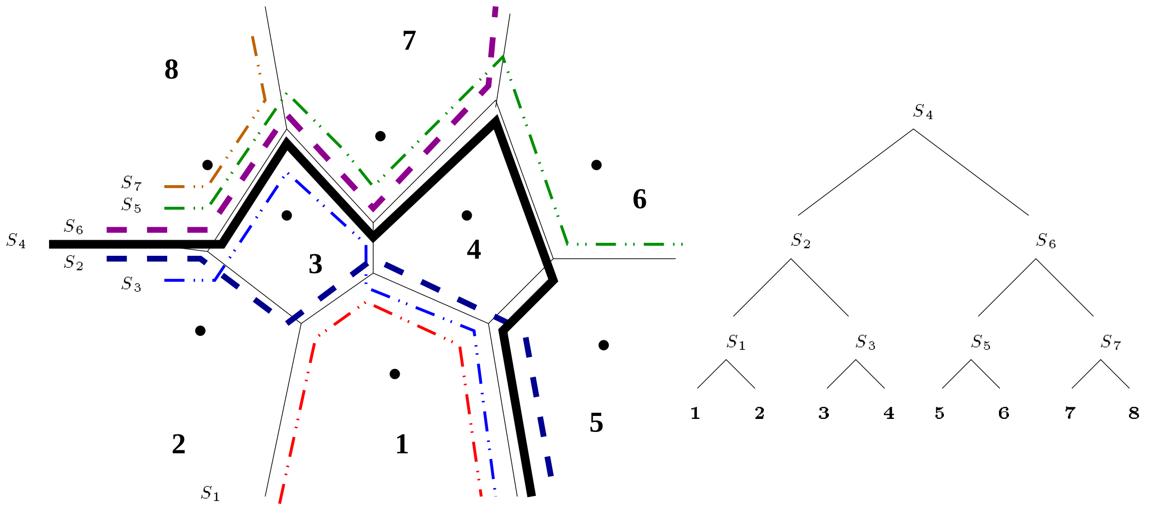

Figure 3: A Voronoi diagram and its separators

The query time can be improved to $\mathcal{O}(\log n)$ by a technique called *fractional cascading*. The intuition behind the technique is that the information that we attained from the binary search at the parent node should help us locate the edge of the separators at the children nodes. This is achieved by linking the edge sets of the parent and the children nodes. A pointer from the edge of the parent node sends us directly to the correct edge to check at the child node. This can be accomplished with only a constant factor increase of the space.

As soon as we leave the two dimensional space, the problem of Nearest Neighbor Search becomes more complex. Even for the 3-dimensional case there is no known solution that is optimal with respect to both space and query time. An attempt to generalize the algorithms for 2 dimensions to the 3-dimensional space stumbles to the fact that the complexity of the Voronoi diagrams in three dimensions is $\Theta(n^2)$. Therefore, any solution that uses Voronoi diagrams in three dimensions is bound to use at least $\Theta(n^2)$ space. There are only few algorithms that tackle directly the NNS problem in three dimensions. The most efficient solution [23], uses a persistent dynamic data structure for planar subdivisions to build a data structure that can be used for point location in space subdivisions. The algorithm requires $\mathcal{O}(n^2)$ space, and has query time $\mathcal{O}(\log^2 n)$.

The best solutions for Nearest Neighbor Search in three dimensions come from the application of algorithms for arbitrary dimensions d to the case $d = 3$. Clarkson [26], and Meiser [66] give algorithms that have optimal query time $\mathcal{O}(\log n)$, but require $\mathcal{O}(n^{2+\delta})$ space. Yao and Yao [89] give an algorithm that achieves linear space, but requires barely sublinear query time. We investigate these algorithms in detail in the following section.

5 Exact Nearest Neighbor Search in d-dimensional Spaces

5.1 Algorithms based on Voronoi diagrams

In this section we present a set of algorithms that tackle the NNS problem by the use of Voronoi diagrams. Some of these algorithms are designed specifically for the problem of point location in Voronoi diagrams, while others solve the more general problem of point location in arrangements of hyperplanes. The algorithms that we described are defined for the real vector space $\mathbb{R}^d$.

5.1.1 The First Attempt

Dobkin and Lipton's seminal paper [33] is the first work to consider the NNS problem for spaces of arbitrary dimension d. The algorithm they propose solves the problem of point location in arrangements of hyperplanes. It uses a simple and intuitive idea, which generalizes the technique described for the case of two dimensions. Given a point location problem in d dimensions, reduce it to one in $d-1$ dimensions. Proceed recursively, until you reach dimension one, where the problem is trivially solved.

Formally, let H_d be a collection of m hyperplanes in $\mathbb{R}^d$, and let $A(H_d)$ be the corresponding arrangement. The pairwise intersections of hyperplanes in H_d produce a set of hyperplanes of lower dimension. Project this set to an arbitrary hyperplane in $\mathbb{R}^d$ which does not belong to H_d (e.g. the hyperplane $x_d = 0$). This produces a new set of hyperplanes H_{d-1} in a space of dimension $d-1$. The number of hyperplanes in H_{d-1} is $\mathcal{O}(m^2)$. Let $A(H_{d-1})$ be the corresponding arrangement. A cell of this arrangement defines a "slab" in the d-dimensional space in a similar way that an interval on the x-axis defines a slab in the 2-dimensional space. The hyperplanes of H_d that intersect this slab do not intersect each other within the slab; therefore, we can order them, and store them in a structure that supports logarithmic search time. Since each slab can contain at most m hyperplanes, if we identify the slab that contains the query point in H_{d-1}, we can locate the point in H_d in $\log m$ time. Therefore, the time $t(m,d)$ to locate a query point q in H_d can be defined recursively as follows:

$$t(m,d) \leq t(m^2, d-1) + \log m.$$

The required space $s(m,d)$ can also be defined recursively:

$$s(m,d) \leq m \cdot s(m^2, d-1).$$

The base cases are $s(m,1) = m$, and $t(m,1) = \log m$. Solving the recurrences we get

$$t(m,d) \leq 2^d \log m \ , \ s(m,d) \leq m^{2^d}.$$

In the case of Nearest Neighbor Search, the set of hyperplanes H_d is generated when we create the Voronoi diagram of the data points. Given a set P of n points, the Voronoi diagram generates $\mathcal{O}(n^2)$ hyperplanes. Therefore, the algorithm of Dobkin and Lipton requires $\mathcal{O}(n^{2^{d+1}})$ space, and has query time $\mathcal{O}(2^{d+1}\log n)$.

5.1.2 Random Sampling Algorithms

In this section we present a technique called *random sampling*, and we see how it can be combined with divide and conquer techniques, to produce algorithms for the NNS problem. The technique is applied to the problem of point location in both Voronoi diagrams and arrangements of hyperplanes.

The input to the algorithm is a set N of n geometric objects, which may be either a set of sites P, or a set of hyperplanes H. Let $A(N)$ denote the arrangement defined by these objects. The general framework of a random sampling algorithm is the following.

- Take a random sample S of the set N, of large enough size s.
- Construct the arrangement $A(S)$, and a data structure for point location in $A(S)$. Since the size of S is small the structure is easy to construct.
- The arrangement $A(S)$ decomposes $\mathbb{R}^d$ into a set of regions $\mathcal{R}_S$. For every region $R \in \mathcal{R}_S$ identify the set of objects in N that conflict with R. Let $N(R)$ be this set.
- For every region R proceed recursively to construct a search structure for the conflict set $N(R)$. The recursion stops when the size of $N(R)$ drops below some fixed threshold.

The resulting data structure is a tree. Each node of the tree corresponds to a region R of $\mathbb{R}^d$, and the set of objects in $N(R)$. The root of the tree corresponds to the whole space $\mathbb{R}^d$, and the set of objects in N. Each node has $|\mathcal{R}_S|$ children, one for each region in $\mathcal{R}_S$. Furthermore, each node is associated with a search structure for identifying the region in $A(S)$ that contains a query point.

Given a query point q, a query on $A(N)$ is answered by descending down the tree. At any node of the tree, we locate the query point in the arrangement $A(S)$. Let R be the region in $A(S)$ that contains the point q. The algorithm descends to the corresponding node, and recursively answers the query over the set of objects in $N(R)$. The query time is equal to the depth of the recursive data structure, times the search time of the search structure for $A(S)$.

We will show that we can force the depth of the search tree to be logarithmic in n, by forcing the size of the conflict sets to be a constant fraction of the size of N. The tools for the proof can be found in the theory of range spaces (for a short introduction into range spaces see appendix A). For every region R in $\mathcal{R}_S$ the conflict set $N(R)$ can be expressed as the intersection of N with a range from a range space with bounded VC-dimension. We can then show that, for some constant α, we can find a random sample S with size independent of n, such that with high probability, S is an α-net for the set N, and each region in $\mathcal{R}_S$ conflicts with at most $\alpha \cdot n$ objects from N (see Theorem 3, appendix A).

There is a critical point in the above discussion. We require that the abovementioned range space has *bounded* VC-dimension, that is, independent of the sample size s. It can be proven that in order to enforce this property we need to guarantee that the complexity (i.e. the number of faces) of each region in $\mathcal{R}_S$ is independent of s. The natural partition of $A(S)$ into cells does not offer this guarantee: we can construct cases where the complexity of some cells is $\mathcal{O}(s)$. We remove this dependency by *triangulating* the arrangement $A(S)$. We define a *triangulation* of a convex polytope as a refinement of the polytope into a collection of simplices whose vertices are also vertices of the convex polytope. A triangulation $\Delta(A(S))$ of the arrangement $A(S)$ is constructed by triangulating the cells of the arrangement. Note that a simplex is a convex polytope on d vertices; its complexity does not depend on the sample size. We define the set of regions $\mathcal{R}_S$ to be the set of all simplices in $\Delta(A(S))$.

The triangulation we consider, which is usually referred to as the *canonical triangulation*, is defined as follows [70]. First assume that the arrangement is restricted in a simplex Γ in $\mathbb{R}^d$. We may think of the vertices of Γ as lying at infinity. We triangulate the faces of the arrangement $A(N)$ inductively on their dimension. For dimension one, we don't need to do anything. Consider now any face f of dimension $j > 1$. Let v be its bottom, that is, the vertex with the minimum x_d-coordinate. By induction hypothesis, all subfaces of f have been triangulated. We triangulate the face f by extending all simplices on its boundary to cones with apex v. The cones are cylinders if the vertex v lies at infinity. It is proven [26] that the triangulation of a convex polytope with n facets has $\mathcal{O}^*(n^{\lfloor d/2 \rfloor})$ simplices [2]. The triangulation of a Voronoi diagram has $\mathcal{O}^*(n^{\lceil d/2 \rceil})$ simplices, and the triangulation of an arrangement of hyperplanes has $\mathcal{O}^*(n^d)$ simplices.

Finally, it is worth noting that the algorithms using random sampling are randomized in a Las Vegas sense; that is, we can repeat random sampling enough times until the resulting data structure has the required properties.

The Clarkson Algorithm

Clarkson [26] was the first to use the idea of random sampling for the NNS problem. His algorithm uses random sampling over the sites in P. The set N of the geometric objects is the set of sites P. The arrangement $A(N)$ is the Voronoi diagram $\mathrm{Vor}(P)$. At each recursive step the algorithm takes a random sample S of size s of the sites in P, and builds the Voronoi diagram $\mathrm{Vor}(S)$. We construct the triangulation $\Delta(\mathrm{Vor}(S))$ of the Voronoi diagram, and we define $\mathcal{R}_S$ to be the set of simplices in $\Delta(\mathrm{Vor}(S))$. The size of $\mathcal{R}_S$ is $\mathcal{O}^*(s^{\lceil d/2 \rceil})$.

For some site $p \in S$, let R be a simplex in the triangulated Voronoi cell $\Delta(\mathcal{V}(p))$. We define the conflict set $N(R)$ to be the set of all sites in P that are closer to some point in R than the site p. Formally, let a be a point in R. The *candidate ball* for the point a, $C(a)$, is the set of points in $\mathbb{R}^d$ that are closer to a than the site p. This area is the open ball centered at a with radius the distance $d(p, a)$. The *candidate region* $C(R)$ for the simplex R is the set of all points in $\mathbb{R}^d$ that are closer to some point in R than the site p. If $v_1, v_2, ..., v_{d+1}$ are the $d+1$ vertices of the simplex R, then it is proven that the candidate region $C(R)$ is equal to the union of the candidate balls centered at $v_1, v_2, ..., v_{d+1}$. We define

[2] Recall that the $\mathcal{O}^*(\cdot)$ notation denotes hidden factors that grow at least as fast as 2^d.

$N(R) = C(R) \cap P$.

Consider the range space $X = (\mathbb{R}^d, B_{d+1})$, where an element B of B_{d+1} is the union of $d+1$ open balls centered at some point in $\mathbb{R}^d$. The set B_{d+1} contains all possible such Bs It is not hard to see that X has VC-dimension $\mathcal{O}(d^2 \log d)$ (see appendix A). Furthermore, no point from S intersects with any of the conflict sets $N(R)$, so we can apply Theorem 3 for ε-nets. Following the general random sampling paradigm we can recursively build a search tree for the Voronoi diagram $\mathrm{Vor}(P)$. Given a query point q, at each node of the tree we identify the simplex that contains the point q by performing a simple brute force search over the $\mathcal{O}(s^{\lceil d/2 \rceil})$ simplices in $\mathcal{R}_S$. The query time of the algorithm is $\mathcal{O}(s^{\lceil d/2 \rceil} \log n)$, and the required space is $\mathcal{O}^*(n^{\lceil d/2 \rceil (1+\varepsilon)})$.

The Meiser Algorithm

Meiser [66] refines the idea of Clarkson by improving the query time of the algorithm. The set N is a set of m hyperplanes, and $A(N)$ denotes the arrangement of the hyperplanes. The algorithm takes a random sample S of N, and builds the arrangement $A(S)$. We construct the triangulation $\Delta(A(S))$ of the arrangement. We define $\mathcal{R}_S$ to be the set of simplices in the triangulation $\Delta(A(S))$. Let R be some simplex in $\mathcal{R}_S$, and let $N(R)$ be the set of hyperplanes that intersect R. It is not hard to show that the set $N(R)$ can be expressed as the intersection of N with a range from a range space with VC-dimension $\mathcal{O}(d^2 \log d)$. Note that no hyperplane in S intersects with the simplex R, so we can apply the Theorem 3 for ε-nets. Therefore, we can apply the general random sampling paradigm, and construct a search tree for $A(N)$.

The main contribution of the paper is that it gives a faster algorithm for searching the simplices at each node, instead of the brute force search that was used by Clarkson. Each face of $A(S)$ is characterized by an s-dimensional vector on the alphabet $\{-, 0, +\}$, depending on its relative position to each hyperplane. At each node, a trie is constructed to keep the faces of the arrangement $A(S)$. Given a query point q, we can locate the face of the arrangement $A(S)$ that contains the query point q, by walking down the trie. Let f be that face. We now need to identify the simplex in f that contains the point q. Let v be the bottom vertex used to triangulate the face f. Let p be the intersection point of the ray $\vec{vq}$ with the boundary of the face f. Note that the ray $\vec{vq}$ runs completely inside the simplex t that contains q. Therefore, the point p is contained in a simplex t' of the subface of f that was used for the construction of simplex t. We are now faced with a new problem of one dimension smaller; locating point p in the subface of f. This is decided in the same way recursively. The recursion terminates when we reach a face of dimension one. Using a small amount of additional pointers we can trace the recursion back to the d-dimensional simplex that contains q. The time for the search algorithm is proven to be $\mathcal{O}(d^5 \log m)$, and the total space required for the data structure is $\mathcal{O}^*(m^{d+\delta})$.

The space requirement can be improved to $\mathcal{O}^*(n^{\lceil n/2 \rceil (1+\varepsilon)})$ for the case of point location in a Voronoi diagram of n sites, by combining Clarkson's random sampling with Meiser's searching technique. The resulting algorithm uses $\mathcal{O}^*(n^{\lceil n/2\rceil(1+\varepsilon)})$ space, and has $\mathcal{O}(d^5 \log n)$ search time.

The Agarwal and Matušek Algorithm

Agarwal and Matušek [1] view the NNS problem as a special case of ray shooting in a convex polytope. Recall that there is a simple transformation from a d-dimensional Voronoi diagram to a $(d+1)$-dimensional upper convex polytope. The problem of finding the nearest neighbor to a query point q reduces to the problem of identifying the first hyperplane on the boundary of the convex polytope that is hit by a ray emanating from infinity and passing through the point q. Agarwal and Matušek solve the problem using a technique called *parametric search* [65, 70].

Parametric search is a powerful technique that reduces search problems to decision problems. We present the technique through its application to the vertical ray shooting problem in appendix B. The parametric search technique reduces the ray shooting problem to the *empty intersection* problem (see appendix B), which is in turn reduced [1] to the *half-space emptiness* problem; given a set of points, and a hyperplane, decide if there is a database point in the half-space defined by the hyperplane. Matušek [64] proves that the half-space emptiness queries can be answered in time $\mathcal{O}^*(\frac{n}{m^{1/\lfloor d/2\rfloor}} \log n)$, using $\mathcal{O}^*(m^{1+\delta})$ space, where $n \leq m \leq n^{\lfloor d/2\rfloor}$. The algorithm is based on results by Chazelle et al. [24] on random sampling. A combination of this result with the parametric search technique gives an algorithm for ray shooting queries, and thus for nearest neighbor queries, that uses $\mathcal{O}^*(m^{1+\delta})$ space, and has query time $\mathcal{O}^*(\frac{n}{m^{1/\lfloor d/2\rfloor}} \log^3 n)$, where $n \leq m \leq n^{\lfloor d/2\rfloor}$.

5.2 Space Partition Algorithms

Space partition techniques have been used extensively in the field of data structures. The underlying idea is to build a hierarchical data structure by recursively subdividing the space into increasingly finer partitions. We divide the space partition algorithms into two categories: those that partition the embedding space from which the points are drawn, and those that partition the data set to be stored. An example of the former are the *region quadtrees* [77] which recursively partition the space into 2^d equal hyper-rectangles. An example of the latter are the *optimized k-d-trees* [47] which recursively partition the data set into two almost equal sets. A survey of such data structures can be found in [78]. More recent results are reported in [48].

We now describe the optimized k-d-trees, a prominent data structure that has motivated a number of similar structures. The k-d-tree is a binary search tree. At each node of the tree the data set is split along some selected dimension into two almost equal subsets, by means of an axes-parallel hyperplane. The dimension to be split is chosen so as to maximize the variance of the points along the corresponding axis. The splitting hyperplane passes through the median of the points along this dimension. We proceed recursively until some threshold condition is reached for the leaves of the tree.

Given a query point q, the search algorithm locates the leaf of the tree that contains the query point, and finds a tentative nearest neighbor among the points that are associated with the leaf node. The algorithm visits all the leafs that intersect the ball B centered at q with radius the distance to the closest point seen so far. In the worst case the algorithm will search the whole tree, resulting in $\mathcal{O}(dn)$ query time. The authors prove that under some assumptions on the distribution of the data and the query points the average query time is $\mathcal{O}(d \log n)$. The space required by the tree is $\mathcal{O}(dn)$.

The idea of space partition is used by Yao and Yao [89] to tackle a more general problem. Their algorithm is designed for *generic* geometric queries; a generic geometric query is defined as the semi-group sum $\sum_{x \in P} Q(x,q)$, where P is the data base, and q is the query point. The function $Q(x,q)$ is defined as one of k bilinear functions $f_i(x,q)$, $1 \leq i \leq k$, that take semi-group values, depending on which one of k bilinear constraints $p_i(x,q)$, $1 \leq i \leq k$, is satisfied. Nearest neighbor queries can be expressed as a special case of the generic queries, where the semi-group is chosen to be $S = (\mathbb{R}, \min)$.

The algorithm is based on the fact that there exists a partition of $\mathbb{R}^d$ into 2^d distinct regions each having at most $n/2^d$ points. The partition is based on the existence of a *projective center*, and it is called *projective partition.* Partitioning each of the subregions recursively we can construct a data structure of size $\mathcal{O}(n)$ for searching the data base. The data structure is a tree called the *partition tree*, where each node of the tree corresponds to some area of the space, and has 2^d children. An important property of the projective partition is that any hyperplane intersects at most $2^d - 1$ of the subregions of the partition. If c is the projective center of the partition, there exists some region of the partition such that for each point x of this region $f_i(x,q) > f_i(c,q)$. Since we want to minimize the function $f_i(x,q)$ we we can always discard this region; therefore at least one of the children of every node, can be discarded when searching the projective tree. This results in slightly sublinear search time $\mathcal{O}(n^{g(d)})$, where $g(d) = \frac{\log(2^d-1)}{d}$.

5.3 Summary of the results

We summarize the results for exact NNS in the following table.

Author(s)	Technique	Objects	Space	Time
Dobkin and Lipton [33]	Projections	Hyperplanes	$\mathcal{O}(n^{2^{d+1}})$	$\mathcal{O}(2^{d+1} \log n)$
Clarkson [26]	Random Sampling	Voronoi diagram	$\mathcal{O}^*(n^{\lceil d/2 \rceil(1+\delta)})$	$\mathcal{O}^*(\log n)$
Meiser [66]	Random Sampling	Hyperplanes	$\mathcal{O}^*(n^{\lceil d/2 \rceil(1+\delta)})$	$\mathcal{O}(d^5 \log n)$
Agarwal and Matušek [1]	Parametric Search, Random Sampling	Voronoi diagrams	$\mathcal{O}^*(m^{1+\delta})$, $n \leq m \leq n^{\lfloor d/2 \rfloor}$	$\mathcal{O}^*(\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log^3 n)$, $n \leq m \leq n^{\lfloor d/2 \rfloor}$
Friedman et al. [47]	Space Partitioning	Points	$\mathcal{O}(dn)$	$\mathcal{O}(dn)$
Yao and Yao [89]	Space Partitioning	Points	$\mathcal{O}(dn)$	$\mathcal{O}(dn^{1-\delta})$

The $\mathcal{O}^*(\cdot)$ notation denotes hidden factors that grow at least as fast as 2^d.

From the above table, we observe that all algorithms that use Voronoi diagrams (or arrangements of hyperplanes) and random sampling have space complexity exponential on the dimension. This seems natural since the complexity of Voronoi diagram is exponential on d, but it is not necessarily an inherent characteristic of Voronoi diagrams, since $\mathcal{O}(n^2)$ hyperplanes are enough to specify a Voronoi diagram. One could hope that we could improve upon the above results. However, current techniques do not seem likely to produce better solutions, since random sampling algorithms rely heavily on the triangulation of the Voronoi diagrams, which imposes space complexity exponential on d. We could hope instead to generalize the tradeoff proven by Agarwal and Matušek [1] for the general case of random sampling algorithms.

6 Approximate Nearest Neighbor Search in d-dimensional Spaces

6.1 The Early Attempts

Arya and Mount [5] were the first to consider the approximate Nearest Neighbor Problem. Their algorithm constructs for each point $p \in P$ a set of cones with angular diameter [3] $\mathcal{O}(\varepsilon)$ that share a common apex p, and cover the space $\mathbb{R}^d$. The number of such cones that cover $\mathbb{R}^d$ is $\mathcal{O}(\varepsilon^{-(d-1)})$. The data structure consists of a directed graph that is called the randomized *neighborhood graph* $NG(P)$, and it is constructed as follows. For every point p, take a random permutation of all points in $P - \{p\}$. For each cone $c(p)$, put an edge from p to some point s in $c(p)$, if s is closer to p than all other points in $c(p)$ that have smaller index. It is proven that the expected number of points sampled from each cone is $\mathcal{O}(\log n)$. Therefore, the expected size of the data structure is $\mathcal{O}(\varepsilon^{-(d-1)} n \log n)$. Given a query point q the search algorithm starts from an arbitrary point p, and it moves progressively towards a point that answers the query, following the edges of the graph. Assume that p is not an $(1+\varepsilon)$-approximate nearest neighbor of the point q, and let $Cl(p)$ is the set of points closer to q than p. The algorithm relies on the idea that we can find a path of edges in $NG(P)$ from the point p to a point s in $Cl(p)$ such that the expected size of $Cl(s)$ is a constant fraction of the size of $Cl(p)$. The expected length of the path is is $\mathcal{O}(\log n)$, and it can be found in $\mathcal{O}(\varepsilon^{-(d-1)} \log^2 n)$ time. Therefore, after $\mathcal{O}(\log n)$ operations we will reach an $(1+\varepsilon)$-approximate nearest neighbor of q. The query time of the algorithm is $\mathcal{O}(\varepsilon^{-(d-1)} \log^3 n)$.

The prohibitive constant factor $\mathcal{O}(\varepsilon^{-(d-1)})$ is improved to $\mathcal{O}(\frac{1}{\varepsilon})^{(d-1)/2}$ by Clarkson [27]. Following the ideas of Arya and Mount, Clarkson constructs for every site p in P a "neighborhood" set N_p with the property that if p is not the $(1+\varepsilon)$-approximate nearest neighbor of some point q, then there exists some point in N_p that is closer to q than p. Starting from an arbitrary point in P the algorithm moves progressively closer to the query point q, until an $(1+\varepsilon)$-approximate neighbor is found. The contribution of Clarkson is that the size of the set N_p is reduced to $\mathcal{O}(dc \log c)$, where c is $\mathcal{O}(\frac{1}{\varepsilon})^{(d-1)/2} d \log(\rho/\varepsilon)$, and

[3]The angular diameter of a cone is defined as the supremum over all angles formed by any two vectors in the cone.

ρ is the ratio of the distance from p to the farthest Delaunay neighbor [4] over the distance to the closest Delaunay neighbor. The data structure for the set of sites P requires $\mathcal{O}(n \log \rho) \times \mathcal{O}(\frac{1}{\varepsilon})^{(d-1)/2}$ space, and has query time $\mathcal{O}(\frac{1}{\varepsilon})^{(d-1)/2} \times \mathcal{O}(\log n)$. Note however that the ratio ρ may become arbitrarily large.

Arya et al. [6] proposed an improved algorithm that is "optimal" for fixed dimensions. The data structure they proposed is based on a hierarchical decomposition of the space, which they call *Balanced Box Decomposition (BBD) tree.* The tree has $\mathcal{O}(\log n)$ height and it recursively subdivides the space into cells. A cell is either a rectangle, or a set theoretic difference of two rectangles one enclosed within the other. Each cell is associated with a set of points that are lying within the cell. Each leaf is associated with a single point. The cells at each level of the tree define a subdivision of the space. The tree has $\mathcal{O}(n)$ number of nodes and it can be created in $\mathcal{O}(dn \log n)$ time.

By the way that the BBD-tree is constructed we can guarantee that the following two degenerate cases do not occur: (a) there are no long and narrow cells, and (b) there cannot exist a series of cells that are nested one within the next, such that they are all extremely close to the boundary of the outer cell. The authors prove that if these two properties hold, then for some radius $r > 0$, the number of cells with diameter at least ℓ that intersect a ball of radius r is at most $\mathcal{O}(dr/\ell)^d$.

The search algorithm works as follows. Given a query point q, we descend down the BBD-tree and find the cell in which the point q is located. Then we visit the leaf cells in order of increasing distance from the point q. Let p be the closest point seen so far. As soon as the distance from q to the next leaf cell in order becomes greater than $d(q,p)/(1+\varepsilon)$, the search terminates. Let r be the distance to the last cell to be visited that did not cause the algorithm to terminate. It is not hard to prove that the diameter of any visited cell is at least εr. Therefore, we obtain a bound $c_{d,\varepsilon} = \mathcal{O}(d/\varepsilon)^d$ on the number of leaf cells that are visited by the search algorithm. Using a priority heap we can locate these points in time $\mathcal{O}(c_{d,\varepsilon} d \log n)$. Therefore, the BBD-tree uses optimal space $\mathcal{O}(dn)$, and has query time $\mathcal{O}(c_{d,\varepsilon} d \log n)$.

Combining ideas from both [5] and [6] Chan [22] provided an algorithm that reduces the exponential factor $\varepsilon^{-(d-1)}$ that appears in the running time of the algorithm in [6] to $\varepsilon^{-(d-1)/2}$. The data structure consists of a set of cones of diameter $\delta = \sqrt{\varepsilon/8}$. For each cone a BBD-tree is constructed that answers a restricted NNS problem, for the points within the cone. The resulting data structure uses $\mathcal{O}(\varepsilon^{-(d-1)/2} n)$ space. The query time of the search algorithm he proposes is $\mathcal{O}(exp(d)(1/\varepsilon)^{(d-1)/2} \log n)$, where $exp(d)$ is a function that grows at least as fast as 2^d. Note that although the query time has been improved the space has increased by a factor of $\mathcal{O}(\varepsilon^{-(d-1)/2})$. Further reduction of the ε dependence by an $\varepsilon^{-1/2}$ factor is possible at the expense of a multiplicative $\log n$ factor in the query time.

Better results can be obtained if one is willing to settle for a rough approximation. Improving upon results by Bern [12], Chan describes an method using quadtrees that finds a $\mathcal{O}(d^{3/2})$-approximate nearest

[4] A Delaunay neighbor of a point p is a point p' such that the Voronoi cells of the two points are adjacent.

neighbor in $\mathcal{O}(d^2 \log n)$ time, using $\mathcal{O}(d^2 n)$ space.

6.2 Recent Algorithms

The algorithms presented so far for the approximate searching problem offer some improvement over the algorithms for the exact search problem since they give optimal solutions for the case of fixed dimensions. However, they still do not remove the curse of dimensionality, since their complexity depends exponentially on d. For some time it seemed probable that the curse of dimensionality carries over to the approximate searching problem. In this section we present some of the more recent algorithms that offer a strong indication that the approximate search problem is easier than the exact.

6.2.1 Randomized Testing

The algorithms that we will present share the common idea of *randomized testing.* A randomized test is defined as a randomized function $\tau : \mathcal{V} \to \mathbb{R}$. Given a query point q, and two points x and y in the database, the randomized test has the property that if $d(y,q) > (1+\varepsilon)d(x,q)$ then the there is a small bias that test τ will favor the point x over y, as the nearest neighbor for q. This bias is amplified by performing a sequence of t such tests, for an appropriately chosen t. This produces a trace $t(x) = \tau_1(x)\tau_2(x)...\tau_t(x)$ for every point x in $\mathcal{V}$. Using the traces of the points in the database we can now perform pairwise comparisons between the points in the database. The algorithms that we will present rely on the following two observations: (a) if the traces of the points in P are precomputed then we can compare two points in time $O(t)$, and (b) if the randomized test takes values in a finite subset of $\mathbb{R}$ that has size c, then there are exactly c^t possible traces. The first property is used in one of the algorithms by Kleinberg to construct an elimination tournament that produces an $(1+\varepsilon)$-approximate nearest neighbor. The second property is used to precompute comparisons between the traces of the points in P, and the set of all possible traces. Given a query point q, we use the precomputed information for the trace $t(q)$ to answer the query.

6.2.2 The Kleinberg algorithms

Kleinberg [59] was the first to obtain a result that improves asymptotically over the simple linear search. He presents two algorithms. The first answers queries in time $\mathcal{O}((d^2 \log d)(d + \log n))$, but requires storage $\mathcal{O}(n \log d)^{2d}$. The second algorithm has query time $\mathcal{O}(n + \log^3 n)$ and requires space $\tilde{\mathcal{O}}(dn)$. The $\mathcal{O}(\cdot)$ notation here hides factors quadratic on ε^{-1}, and the $\tilde{\mathcal{O}}(\cdot)$ notation hides factors that are polynomial in $\log n$ and ε^{-1}. When $d = \mathcal{O}(n/\log^3 n)$ the second algorithm has better query time than the $\mathcal{O}(dn)$ query time of the linear search. The algorithms are defined for the space $\mathbb{R}^d$ under the Euclidean norm.

The two algorithms use the same randomized test which relies on the following simple idea: given a set of points P, and a query point q, if we project all points on a random line that passes through the origin, then there is a small bias that the projection will preserve the relative positions of the points with respect to the query point q. Formally, we define the randomized test as follows. Select a unit vector v from the d-dimensional unit sphere, uniformly at random. For any point x, we define $\tau_v(x) = v \cdot x$.

Let x and y be two points in P, and q a query point. We can prove that if $||y - q|| > (1+\varepsilon)||x - y||$, then $|\tau_v(y) - \tau_v(q)| > |\tau_v(x) - \tau_v(q)|$ with probability at least $\frac{1}{2} + \frac{\varepsilon}{3}$. We call this event the *distinguishing property* of vector v. The vectors with the distinguishing property can be expressed as ranges of a range space with bounded VC-dimension. Taking a sample D of $\Theta(d \log^2 d)$ vectors we obtain a γ-sample with probability $1 - \delta$ (see appendix A). For an appropriate choice of γ, we can prove that if the set D is selected correctly (i.e., if we avoid the event with probability δ), then at least half of the vectors in D have the distinguishing property for all x,y in P. The set D is called a *distinguishing set.*

Given a distinguishing set we can now make comparisons between two points x and y. If $|\tau_v(x) - \tau_v(q)| < |\tau_v(y) - \tau_v(q)|$ for at least half of the vectors in D then $||x - q|| \leq (1+\varepsilon)||y - q||$ (since otherwise by definition of D as a distinguishing set, it should be that $|\tau_v(x) - \tau_v(q)| > |\tau_v(y) - \tau_v(q)|$). We say that x D-dominates y.

The first algorithm restricts the values of the randomized test in a finite set by means of the following transformation. First, we find a distinguishing set D of $t = \Theta(d^2 \log d)$ vectors. Given the set D we project the points in P onto the vectors in D. We also project the middle-points $p_{ij} = \frac{1}{2}(p_i + p_j)$ for every pair p_i, p_j in P. The projections define a set of n^2 intervals for every vector v_ℓ in D. Given a query point q, and a vector v_ℓ in D, we define $\tilde{\tau}_\ell(q)$ to be the interval σ_ℓ on v_ℓ that contains $v_\ell \cdot q$. The trace of the query is defined as $\tilde{t}(q) = \tilde{\tau}_1(q)\tilde{\tau}_2(q)...\tilde{\tau}_t(q)$.

The idea of the algorithm is simple and intuitive. Roughly speaking, we answer a query for a point q with the point whose projection is the closest to the projection of q in the majority of the vectors. However, instead of computing the relative positions of the projected points with respect to the projection of q, we compute the relative positions of the projections with respect to the interval that contains q. Since there is a finite number of such intervals, we can precompute this information in preprocessing time.

For some interval σ_ℓ on a vector v_ℓ, we say that p_i is closer to σ_ℓ than p_j if the projection of the middle-point p_{ij}, lies between the projection p_j and σ_ℓ. For some trace $\sigma = \sigma_1\sigma_2...\sigma_t$ we say that p_i σ-dominates p_j if p_i is closer to σ_ℓ in more than half of the vectors in D. Note that if σ is the trace $t(q)$ of a query point q, then if p_i σ-dominates p_j, then p_i also D-dominates p_j. Therefore, $||p_i - q|| \leq (1+\varepsilon)||p_j - q||$.

The actual data structure is a table that has an entry for every *realizable* trace σ, that is, for every trace for which there is a query point q such that $\sigma = \tilde{t}(q)$. The number of realizable traces is $\mathcal{O}(n \log d)^{2d}$. For every realizable trace σ, we define an ordering (called apex ordering [59]) on the points in P with respect to the σ-dominates relation. We store the first element of the ordering. We can prove that the first point in the apex ordering is an $(1+\varepsilon)$-approximate nearest neighbor for any point q with trace σ.

The searching algorithm simply projects the query point q on every vector v_ℓ in D, and locates the interval in which it falls using binary search. Given the trace of the query point q we look up the table and return the first element of the apex ordering. The data structure requires space $\mathcal{O}(n \log d)^{2d}$, and has query time $\mathcal{O}(d \log^2 d(d+\log n))$. We can easily modify the algorithm to return the k $(1+\varepsilon)$-approximate nearest neighbors, simply by returning the first k elements of the apex ordering.

The second algorithm is less intuitive so we will only give a brief sketch of it. The algorithm builds an elimination tournament on the set of points. The data structure is constructed by placing the points in P at the leaves of a binary tree with depth $\log n$. Let Γ be a distinguishing set of vectors of size $\mathcal{O}(\log^3 n)$. Starting from the leaves of the tree we perform pairwise comparisons all the way up to the root of the tree. The winner of the tournament is an $(1+\varepsilon)$-approximate nearest neighbor with high probability. The proof relies on the following two observations. First, the exact nearest neighbor of a query point q will advance to height $b = \log\log n$ of the tree (the leaves are at height 0) with high probability. Up to this height we perform comparisons using a subset of Γ with constant size. Second, the size of Γ is large enough so that if we perform Γ-comparisons from height b up to the root of the tree the approximation error that is accumulated is at most $(1+\varepsilon)$. Using a matrix M that keeps the precomputed inner products of the vectors in Γ with the points in P, we can achieve query time $\mathcal{O}(n + d\log^3 n)$. The algorithm requires space $\mathcal{O}^*(nd)$, where the $\mathcal{O}^*()$ notation hides factors that are polynomial in $\log n$.

6.2.3 The Hamming Space

The following two algorithms [61, 56] that we present consider the approximate NNS problem for the Hamming space (H_d, h), where H_d is the d-dimensional Hamming cube, and h is the Hamming distance. Then they show that an instance of the problem in the space $(\mathbb{R}^d, L_p)$, where $p = 1, 2$, can be reduced to a small number of instances of the problem in the Hamming space.

We present the two algorithms together, since the technique they employ is essentially the same. The idea is to use the randomized testing technique to solve the following decision problem.

Definition 3 (ε-DNNS$_\ell$(P,q)) *Given a set of points P, and a query point q, for some distance ℓ, $1 \le \ell \le d$,*

- *return* YES *if there is some point $x \in P$ such that $h(x,q) \le \ell$,*
- *return* NO *if for every point $x \in P$ $h(x,q) > (1+\varepsilon)\cdot\ell$,*
- *return either* YES *or* NO *in all other cases.*

We define a randomized test $\tau : H_d \to \{0,1\}$ that has the following property: given a query point q, and a constant ℓ, $1 \leq \ell \leq d$, for any two points x, y in P,

- if $h(x,q) \leq \ell$, then $\text{Prob}[\tau(x) = \tau(q)] \geq \delta_1$,
- if $h(y,q) > (1+\varepsilon)\cdot\ell$, then $\text{Prob}[\tau(y) = \tau(q)] \leq \delta_2$,

with $\delta_1 > \delta_2$. There is a small bias $\delta = \delta_1 - \delta_2 = \mathcal{O}(\varepsilon)$ towards favoring the correct point. We create a trace of t such tests. This is a t-dimensional binary string. We use this string to index a table T of size 2^t, where we store information for every possible trace.

The two algorithms use a different approach at this point. Kushilevitz et al. [61] create a large trace, resulting in large storage space, but small search time, since the search algorithm needs to compute only a single trace. On the other hand, Indyk and Motwani [56] create a small trace, resulting in small storage space, but large search time, since the search algorithm must compute a large number of traces.

Specifically, Kushilevitz et al. [61] prove that if we select $t = \mathcal{O}(\delta^{-2}\log(n\log d))$, we can ensure that if for some point x, $h(t(x), t(q)) \leq (\delta_1 + \frac{1}{3}\delta)t$, then $h(x,q) \leq (1+\varepsilon)\ell$, with high probability. For each entry a of the table T we keep a point x such that $h(t(x), a) \leq (\delta_1 + \frac{1}{3}\delta)t$, if such a point exists. We keep $\mu = \mathcal{O}(d\log d)$ such tables. When answering a query, we select at random some table and look up the trace of the query point $t(q)$. The time for the search is $\mathcal{O}(d\log n + d\log d)$. The size of the data structure however depends exponentially on $\mathcal{O}(\varepsilon^{-2})$; the structure requires $\mathcal{O}(\text{poly}(d)\log d(n\log d)^{\mathcal{O}(\frac{1}{\varepsilon^2})})$ space.

Indyk and Motwani [56], draw the randomized test from a family of *locality sensitive hashing* functions. Locality sensitive hash functions is a set of hash functions such that if we select one at random we obtain a randomized test with the properties we described above. Intuitively, this means that the probability of collision is greater for points that are closer to each other. The algorithm for general families of locality sensitive hashing functions evaluates $t = \mathcal{O}(\log n)$ hash functions, and stores each point x in the entry $t(x)$. We create $\mu = \mathcal{O}(n^\rho)$ tables, where $\rho = \frac{\ln(1/\delta_1)}{\ln(1/\delta_2)}$. Given a query point q we check the entry $t(q)$ in all μ tables, and we return YES if we encounter a point that is within distance $(1+\varepsilon)\ell$, and NO otherwise. For the case of hamming space, this results in an algorithm with storage requirements $\mathcal{O}(dn + n^{1+1/(1+\varepsilon)})$ and query time $\mathcal{O}(dn^{1/(1+\varepsilon)})$.

Having solved the decision problem we can now find an $(1+\varepsilon)$-approximate nearest neighbor, by performing a binary search on all possible distances in $\{1, (1+\varepsilon), (1+\varepsilon)^2 ..., (1+\varepsilon)^{\log_{(1+\varepsilon)} d}\}$. The search algorithm finds a number k, $1 \leq k \leq \log_{(1+\varepsilon)} d$, such that the distance of the exact nearest neighbor is within the interval $((1+\varepsilon)^k, (1+\varepsilon)^{k+1})$. The number k can be found in time $\mathcal{O}(\log\log d)$. Any point that has distance within this interval, can be returned as an $(1+\varepsilon)$-approximate nearest neighbor. Therefore, the search algorithm for the $(1+\varepsilon)$-approximate nearest neighbor adds a multiplicative factor $\mathcal{O}(\log d)$ to the space requirements, and a multiplicative factor $\mathcal{O}(\log\log d)$ to the query time of the decision algorithm.

6.2.4 Approximate Nearest Neighbor Search in the Euclidean space

Kushilevitz et al. [61] reduce the problem of approximate NNS in the Euclidean space to $(1+\varepsilon)$-approximate NNS in a Hamming space of dimension DS, for D and S to be specified later. We select D lines that pass through the origin uniformly at random, and we place S breakpoints on each line, where S is an appropriately selected constant. For every point p we create a DS-dimensional vector $\eta(p)$, such that the bit $\eta_{ji}(p)$ is set to one if the projection of p on the j-th line lies after the i-th breakpoint. We create $\mathcal{O}(n^2)$ such Hamming cubes. The authors prove that if $D = \mathcal{O}(d^2 \log d)$, then the projections preserve the lengths of the vectors on the average. Using this property, we can limit the search to only $\mathcal{O}(\log n)$ Hamming cubes. The resulting data structure uses $\mathcal{O}((nd \log(dn/\varepsilon))^{O(\varepsilon^{-2})})$ space, and has query time $\mathcal{O}(d\text{poly}(\log(dn/\varepsilon)))$, where the $\mathcal{O}(\cdot)$ notation hides factors polynomial in $\frac{1}{\varepsilon}$. For more details, and more insight into the proof see appendix C.

Indyk and Motwani reduce the $(1+\varepsilon)$-approximate NNS problem to a small set of ε-DNNS problems, by means of a structure called the *Ring-Cover tree*. The Ring-Cover tree recursively decomposes the space into a set of clusters with small diameter. We generate an ε-DNNS problem for the points in each cluster. The decomposition has the property that for some cluster C, and some query point q, if the algorithm for the ε-DNNS(C,q) returns NO, then we do not need to search the cluster C any further, since either the points in C are not $(1+\varepsilon)$-neighbors of q, or the cluster C is far enough from q, so that all points are equivalent as $(1+\varepsilon)$-approximate neighbors of q. The Ring-Cover tree generates $\mathcal{O}(n\text{poly}\log n)$ number of ε-DNNS instances, and it can be searched in $\mathcal{O}(\text{poly}\log n)$ time.

The authors claim that they can transform an instance of the problem from the space $(\mathbb{R}^d, L_2)$ to an instance of the problem in the Hamming space (H_d, h). Using the results for the Hamming space they produce an algorithm that uses space $\mathcal{O}(dn + n^{1+1/(1+\varepsilon)})$ and has query time $\mathcal{O}(dn^{1/(1+\varepsilon)})$.

Another approach for solving the ε-DNNS$_r$(P,q) problem in the real space $\mathbb{R}^d$ is to apply the *bucketing method.* Assume that $r = 1$. We impose a grid of spacing $\frac{\varepsilon}{\sqrt{d}}$ on the space $\mathbb{R}^d$. Clearly any two points in a cell have distance at most ε. For any ball $B_i = B(p_i, 1)$ defined by the ε-DNNS problem, let $\overline{B_i}$ denote the set of grid cells that intersect the ball B_i. We store the cells in $\cup_i \overline{B_i}$ into a hash table, together with information about the center point of the ball B_i. Hashing is possible, since the clusters in the Ring-Cover tree, have bounded diameter. Given a query point q we compute the cell that contains q and we hash it into the hash table. If the cell belong to some ball B_i then we return YES, otherwise we return NO. We can prove that each ball intersects with $\mathcal{O}(\frac{1}{\varepsilon})^d$ cells. Therefore, the total space of the algorithm is $\mathcal{O}(n) \times \mathcal{O}(\frac{1}{\varepsilon})^d$ The query is answered in time $\mathcal{O}(d)$, the time to compute the hash function. The authors combine this result with a well known theorem by Frankl and Maehara [42], which states that the projection of the point set P onto a subspace defined by $9\varepsilon^2 \log n$ lines preserves all inter-point distances within a relative error ε. The resulting algorithm uses space $(nd)^{\mathcal{O}(1)}$ and has query time $\mathcal{O}(d\text{poly}\log n)$. The constant $\mathcal{O}(1)$ depends on ε^{-2}.

6.3 Summary of the results

We summarize the results for approximate NNS in the following table.

Author(s)	Space	Storage	Time
Arya and Mount [5]	$(\mathbb{R}^d, L_p)$	$\mathcal{O}(\varepsilon^{-(d-1)} n \log n)$	$\mathcal{O}(\varepsilon^{-(d-1)} \log^3 n)$
Clarkson [27]	$(\mathbb{R}^d, L_p)$	$\mathcal{O}(n \log \rho) \times \mathcal{O}(\frac{1}{\varepsilon})^{-(d-1)/2}$	$\mathcal{O}(\frac{1}{\varepsilon})^{-(d-1)/2} \times \mathcal{O}(\log n)$
Arya et al. [6]	$(\mathbb{R}^d, L_p)$	$\mathcal{O}(dn)$	$\mathcal{O}((\frac{d}{\varepsilon})^d d \log n)$
Chan [22]	$(\mathbb{R}^d, L_p)$	$\mathcal{O}(\varepsilon^{-(d-1)/2} n)$	$\mathcal{O}(\frac{1}{\varepsilon})^{(d-1)/2} \times \mathcal{O}(\log n)$
Kleinberg [59]	$(\mathbb{R}^d, L_2)$	$\mathcal{O}(n \log d)^{2d}$	$\mathcal{O}((d^2 \log d)(d + \log n))$
Kleinberg [59]	$(\mathbb{R}^d, L_2)$	$\tilde{\mathcal{O}}(dn)$	$\mathcal{O}(n + \log^3 n)$
Kushilevitz et al. [61]	(H_d, h)	$\tilde{\mathcal{O}}(nd)^{\mathcal{O}(\varepsilon^{-2})}$	$\mathcal{O}(d \text{poly} \log(dn))$
Indyk and Motwani [56]	(H_d, h)	$\mathcal{O}(dn + n^{1+1/(1+\varepsilon)})$	$\mathcal{O}(dn^{1/(1+\varepsilon)})$
Kushilevitz et al. [61]	$(\mathbb{R}^d, L_{1,2})$	$\tilde{\mathcal{O}}(nd)^{\mathcal{O}(\varepsilon^{-2})}$	$\mathcal{O}(d \text{poly} \log(dn/\varepsilon))$
Indyk and Motwani [56]	$(\mathbb{R}^d, L_{1,2})$	$\mathcal{O}(dn + n^{1+1/(1+\varepsilon)})$	$\mathcal{O}(dn^{1/(1+\varepsilon)})$
Indyk and Motwani [56]	$(\mathbb{R}^d, L_p)$	$\mathcal{O}(n) \times \mathcal{O}(\frac{1}{\varepsilon})^d$	$\mathcal{O}(d)$
Indyk and Motwani [56]	$(\mathbb{R}^d, L_{1,2})$	$\tilde{\mathcal{O}}(nd)^{\mathcal{O}(\varepsilon^{-2})}$	$\mathcal{O}(d \text{poly} \log n)$

The $\mathcal{O}(\cdot)$ notation hides factors polynomial in $\frac{1}{\varepsilon}$. The $\tilde{\mathcal{O}}(\cdot)$ notation hides factors polynomial in $\log n$ and ε^{-1}. The constant ρ is the furthest to shortest Delaunay neighbor distance ratio.

7 A practical approach to Nearest Neighbor Search

The Nearest Neighbor Search problem, apart from being an interesting problem in itself, it is also of great importance in many practical applications. Therefore, there is an increasing interest in creating data structures that can answer Nearest Neighbor queries sufficiently well on the average, or for some restricted data sets. Since in most applications we expect the size of the data set to be very large, it is not likely that the data can fit in main memory, so it would have to be distributed in blocks of the secondary memory. We index the blocks in secondary memory, using specialized data structures, typically called *indexes*. An index is a searchable data structure that keeps pointers to the blocks in the secondary memory that store the actual data. When processing a query we search the index to determine the blocks to be accessed. Following the pointers, we locate and retrieve the blocks from secondary memory. In this context, the dominant factor in the cost of processing a query is the number of blocks that must be accessed in order to answer the query. The space requirements are measured in terms of number of blocks required to store the points of the database, and any additional necessary information. For most practical applications the space required by the index should be linear in the number of points, with a small multiplicative constant.

The problem can be solved optimally for the case of 1-dimensional spaces, using the B-tree and its variants [28]. B-trees impose a hierarchical decomposition of the data. They consist of balanced trees, where each node of the tree corresponds to an interval of points. The children of an internal node are mutually disjoint subsets that partition the interval of the parent. The leaves have pointers to the data in the secondary memory. It can be proven that the number of accessed blocks on average and in the worst case for the B-trees is logarithmic in the size of the database [29, 28].

The problem becomes more complex when we move to spaces of higher dimension. There is a multitude of data structures that deal with the problem of indexing multidimensional spaces. In general, we can classify the indexes in two main categories: *hierarchical* indexes, that have a structure similar to the one of the B-tree (e.g. R-trees, k-d-trees), and *bucket* indexes that use hash-like functions for finding the accessed blocks (e.g. hashing, grid files). A survey on the multidimensional indexing problem can be found in [48]. A more specialized survey on indexes and nearest neighbor queries can be found in [10].

7.1 Algorithms for Nearest Neighbor Search

We now present two algorithms for Nearest Neighbor Search. The algorithms can be applied to any hierarchical index that makes use of what we term *container blocks* [53]. This term is used to denote an area of the space that may itself be decomposed further, depending on the number of points it contains. The resulting index consists of a tree of nested container blocks.

7.1.1 The HS algorithm

The algorithm proposed by Hjaltason and Samet [53] uses a simple idea that can be traced back to a number of different works (e.g. k-d-trees [47], or BBD-trees [6]). Given a query point q, the idea is to search the blocks, in order of increasing distance from the point q. Formally, let MINDIST(q,B) denote the minimum distance from point q to any point in the container block B. MINDIST(q,B) is zero if q is contained within the block B. We think of the function MINDIST as the distance from q to the block B. Note that no database point in B can be closer to q than MINDIST(q,B).

The algorithm searches the hierarchical structure of the index to locate the container block that contains the point q. As we descend down the tree we enter into a priority queue the container blocks that we encounter along the search path. The blocks are sorted according to the MINDIST function. The algorithm then visits the blocks in order of increasing distance from the point q. If the container block corresponds to an internal node of the tree, then we enter all the blocks it contains into the priority queue. If a container block corresponds to a leaf node, then we access the block and we compute the distances from q to all points within the block. We keep the distance to the closest point seen so far in a variable ℓ_{min}. The algorithm terminates when the distance to the next block to be visited becomes greater than ℓ_{min}.

The HS algorithm is optimal, in the sense that it only visits the blocks that intersect with the ball centered at the query point, with radius d_{min}, the distance of the nearest neighbor. In the worst case however, it will access all the blocks of the index.

7.1.2 The RKV algorithm

A variation of the same idea was proposed by Roussopoulos et al. [75]. Given a query point q, the algorithm starts from the root of the tree and performs a depth-first traversal of the tree. At each node the children are sorted according to some criterion, and they are visited in that order. The authors report the MINDIST function as a good sorting criterion. If the next node to be visited is a leaf node then we compute the distance from q to all the points in the block. The distance of the closest point seen so far is stored in the variable ℓ_{min}. A node of the tree is not visited (we say that it is *pruned*) if its MINDIST is greater than ℓ_{min}.

The main difference between the two algorithms is that the RKV algorithm fully explores a branch of the tree before moving on to the next, while the HS algorithm jumps between the different branches of the hierarchy. As a result, the RKV algorithm may access more than the “optimal” number of blocks. (see figure 4)

A refinement of the algorithm is defined in the case of indexes that use *Minimum Bounding Rectangles* (MBRs) [5]. By definition, every face of an MBR contains at least one point of the database. Let f be a face of the MBR B, and let p be a database point on f. The maximum distance, $\ell_{max}(q,f)$, from q to any point on f gives an upper bound on the distance from q to p. We define MINMAXDIST(q,B) to be the minimum over all faces of the MBR B. Note that the distance from q to the closest point in B cannot be greater than MINMAXDIST(q,B). (see figure 4)

We use the above observation to prune the branches of the tree as we descend down the hierarchy. Specifically, if MINDIST(q,B) of an MBR B is greater than MINMAXDIST(q,B') of some MBR B', then B is discarded since it cannot contain the nearest neighbor. Similarly, if the tentative nearest neighbor has ℓ_{min} greater than MINDIST(q,B) for some MBR B, then it is discarded since it cannot be the nearest neighbor. The MINMAXDIST can also be used as the sorting criterion for the search. There are cases that this reduces the number of blocks accessed, but in general the authors report MINDIST as a better sorting criterion.

The new pruning rules offer some improvement over the simple algorithm when we consider on indexes that use MBRs. We still expect however the HS algorithm to perform better, since it accesses less blocks. Nevertheless, Berchtold et al. [10] argue that in practice there are cases that the RKV algorithm performs better because it accesses the blocks in the order in which they are stored in the

[5] Given a set of points S, we define the MBR for the points in S as the rectangle with the minimum volume that contains all points in S

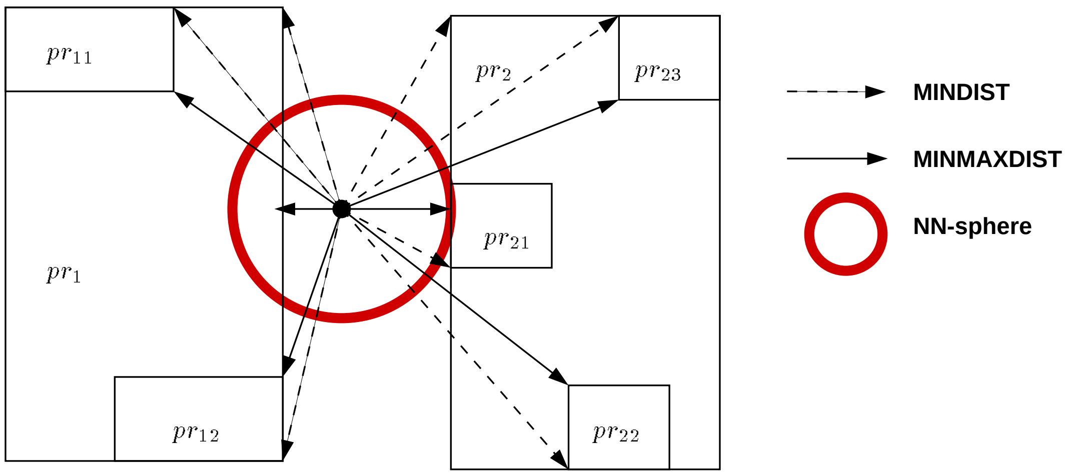

Figure 4: MINDIST and MINMAXDIST. The schedule of the HS and RKV algorithms

secondary memory.

7.2 Indexes for Nearest Neighbor queries

The R-tree [51] is probably the index most commonly used in practice. The R-tree is a direct generalization of the B-tree for spaces of dimension greater than one. It consists of a tree of nested Minimum Bounding Rectangles (MBRs). A leaf node consists of the MBR of the points stored in a disk block. An internal node is the MBR of the MBRs of its children. When a new point is inserted that causes an MBR to overflow, we split the MBR into two MBRs. The disadvantage of splits is that they may cause the overlap between different MBRs, which results in higher search time. The R*-tree [7] addresses this problem by forcing the points in the MBR to be reinserted, resulting in fewer splits, and less overlap.

Despite their popularity, R-trees are not well suited for nearest neighbor queries in high dimensions, since they are designed with respect to the optimization of range queries. Recently, there has been an increasing interest in the design of specialized indexes for nearest neighbor queries. We have already seen some data structures such as the BBD-trees [6], or the space partition methods [47, 89], that can be easily adapted to work as indexes for secondary memory. In this section we present some of the indexes designed for indexing multidimensional data for nearest neighbor queries.

The SS-tree

The SS-tree [87] is inspired by the R*-tree, but instead of bounding hyper-rectangles it uses bounding spheres. Given a set of points it encloses them in a sphere that is centered at the centroid of the points, and has radius the largest distance to any of the points. In the case of an overflow, the SS-tree applies a similar algorithm to the R*-tree with forced reinsertions. When a sphere must be split, we split it into two spheres, such that each sphere has approximately half of the points. Experimental observations show that bounding spheres have shorter diameter than the bounding rectangles on the average. This favors the nearest neighbor queries, since each node contains points that are close together. On the other hand, bounding spheres have greater volume, which increases the overlap between different spheres. The authors implement a variant of the RKV algorithm, and they show that the SS-tree outperforms the R*-tree.

The SR-tree

The SS-tree has the advantage that it groups the points into regions that have short diameter, but they exhibit high overlap. On the other hand, the R*-tree groups the points into regions that have small overlap, but their diameter increases exponentially with the dimension. The SR-tree [58] attempts to combine the advantages, and overcome the shortcomings of both these methods, by using both bounding spheres and bounding rectangles. A region in the SR-tree is defined as the intersection of the bounding sphere (as it is defined for the SS-tree), and the MBR that contains the points in the sphere. The authors implement a variant of the RKV algorithm for nearest neighbor queries. They show that the SR-tree outperforms both the SS-tree, and the R*-tree.

The X-tree

The X-tree [11] is an index designed for multidimensional data. The X-tree relies on the observation that as the dimension increases the overlap between the bounding rectangles also increases. If the overlap is high then a linear scan of the data is better. The reason is that due to the high overlap almost the whole data file must be searched. In this case a linear scan is faster, because it accesses the blocks in the order in which they are stored in disk. The underlying idea is to combine these two approaches into one hybrid structure. When we have to split a rectangle, we should split it along a dimension that minimizes the resulting overlap. If no such dimension exists, then we do not split the rectangle. Instead, we create a super-node that contains all the points of the rectangle. The X-tree uses the RKV algorithm to answer the nearest neighbor queries. The authors show that it outperforms the R*-tree. Furthermore, they claim that it outperforms the SS-tree and the SR-tree, by comparing their improvement relative to the R*-tree. One caveat in the above approach is that the design of the X-tree is based on the assumption that data and query points are uniformly distributed.

The VP-tree

The VP-trees [82, 90, 16] use a simple and intuitive idea. Given a set of points P, we select a point p to be the *vantage point*. We compute the distance from p to all other points in P, and we use these distances to compare the points. Formally, for some point $x \in P$, let $\Pi_p(x) = d(p, x)$. Given two points x and y in P, we define the distance between x and y relative to the vantage point p as $d_p(x, y) = |\Pi_p(x) - \Pi_p(y)|$. Compute the value $\Pi_p(x)$ for each point $x \in P$. Let μ denote the median of the distances $\Pi_p(x)$, for $x \in P$. The median μ creates a natural partition of the points into two approximately equal sets P_L and P_R: the points that have $\Pi_p(x) < \mu$, and those that have $\Pi_p(x) > \mu$ respectively. The construction of the tree proceeds recursively, defining a vantage point for each of the subsets P_L and P_R, until their size becomes equal to the bucket size.

Consider now, the NNS(τ) problem, where we want to return a nearest neighbor of a query point q, if it lies within distance τ. Note that for every point $x \in P$, $d(q,x) \geq d_p(q,x)$. It is not hard to see that if $\Pi_p(q) \geq \mu + \tau$, then the nearest neighbor cannot belong in the set P_L. We can therefore, restrict the search to the set P_R. Similarly, if $\Pi_p(q) \leq \mu - \tau$, the nearest neighbor cannot belong in the set P_R, and we can restrict the search to the set P_L. In all other cases, we search both P_L and P_R. Unfortunately, there is no comparative study of the VP-trees with other indexes, with respect to block accesses. An interesting probabilistic analysis under the homogeneity assumption can be found in [25].

Density based indexing

In some applications such as pattern matching we can think of the database and the query points as being generated by a mixture of distributions. The idea behind density based indexing [8] is to use this fact when constructing an index for nearest neighbor queries. Using EM (Expectation Maximization) techniques we produce a probability distribution that best fits the data, which consists of a mixture of Gaussians. For every point in the database we assign it to one of the clusters defined by the mixture of Gaussians, using the optimal Bayes decision rule. We built an index for each cluster. Given a query point q, we initially assign it to some cluster. We search the index for that cluster, and find a nearest neighbor within that cluster. We then estimate the probability that the nearest neighbor belongs to a different cluster. If this probability exceeds some tolerance, we proceed searching other clusters.

The authors perform experiments on both synthetic, and real data. For data generated by a mixture of Gaussians, the algorithm performs very closely to the ideal algorithm that scans the minimum number of clusters. The authors encounter some technical problems when testing on real data, so they report a small number of experiments. In these tests, their algorithm accesses approximately twice the amount of data accessed by the ideal algorithm.

Hashing

Hashing techniques produce efficient indexes for exact match queries. Nearest neighbor queries are harder, since hashing functions are usually designed so as to scatter the points in space, not to cluster them. What is needed is hashing functions that distribute the points evenly into buckets, and preserve the locality in space at the same time.

Locality preserving hash functions is one such case. Given a hash function h, and some constant c, we say that h is *c-expansive*, if for any two points q and p, $d(h(p),h(q)) \leq d(p,q) + c$. For $c = 0$, we say that the function h is *non-expansive*. Linial and Sasson [62] show that there exist non-expansive hash functions for the 1-dimensional space. Indyk et al. [57] use these functions to construct $\sqrt{d}$-expansive hash functions for the d-dimensional space under the L_2 norm. They show that the bucket size is bounded by a constant B, with constant probability (but not arbitrarily small). Unfortunately, the constant B depends exponentially on d. Locality preserving hash functions can be used for the λ-neighborhood problem: given a query point q, return all points within radius λ from q. The algorithm needs to search only the buckets that are within distance $\lambda + c$ from the bucket $h(q)$ that the query point is hashed to.

We have already presented *locality sensitive* hash functions. The idea in this case is that points that are close in space have higher probability to be hashed to the same bucket. Locality sensitive hash functions are used to answer approximate nearest neighbor queries. Gionis et al. [50] propose an implementation of the algorithms described in [56], and they show that locality sensitive hashing outperforms the SR-trees.

The AV-file

Recent studies [86] have shown that under the uniformity and independence assumption for the data, as the dimension grows the performance of partitioning index methods degrades to that of a linear scan. The AV-file [86, 85] provides a simple method to accelerate this linear scan. Assuming that the data space is the unit cube, we impose a grid upon the space. For every point in the database the AV-file stores the binary representation of the cell that contains it. Given a query point q, we can derive upper and lower bounds to the distance of the point q to all points in the database. Points with lower bound greater than the upper bound of some other point can be immediately discarded. We form a candidate set of nearest neighbors for q, and we access the actual vectors at the secondary memory in order of increasing distance from q. The authors find analytical formulas in the case that the data is uniformly and independently distributed. Furthermore, they show that their method improves upon R-trees and X-trees in the case of high dimension. However, their method relies heavily on the assumption of uniformity and independence, and it is not clear how it performs when the AV-file cannot fit in main memory.

7.3 Evaluation of indexes

It is not hard to see that for all the above indexes we can construct a worst case scenario that will force the search algorithm to access all the blocks in the secondary memory. The worst case for most indexes is the case that all points are placed close to the surface of a sphere, and the query point is chosen to be the center of the sphere. In this case all database points seem almost equivalent to the query point. Since worst case upper bounds do not help to discriminate between different indexes, experimental evidence is usually used in order to proclaim the superiority of one index over another. Indexes are tested on real data, as well as on synthetic data, generated from some distribution. A commonly used assumption is that the data and the query points are uniformly and independently distributed. This is a simplifying assumption rather than a realistic one; real data is usually clustered and highly correlated.

Nevertheless, even under the assumption of uniformity and independence, we are still unable to extract good guarantees on the average performance of the indexes. Weber et al. [86] show that for large dimension the performance of indexes that perform any sort of partition is no better than a linear scan of the data file. This is a result of the fact that as the dimension increases the data becomes more sparse, and the expected distance of the query to the nearest neighbor (denoted by d_{min}) grows with the dimension. Even if we use an "optimal" algorithm that accesses only the blocks that intersect the ball around the query point with radius d_{min} all blocks must be accessed on average. In fact, under the above assumptions, it is shown both experimentally and analytically that for large dimension (but not exceedingly large) linear scan outperforms all indexing methods; a linear scan accesses the pages in the order in which they are arranged in the disk, which is approximately ten times faster than an unordered retrieval of the same pages. The same conclusion is reached in a more careful analysis by Berchtold et al. [9] that takes into account the boundary effects. Therefore, when evaluating an index method it is also important to compare it with the linear scan.

Results of similar flavor are obtained by Beyer et al [14]. They show that under certain broad conditions (which include all i.i.d. distributions) when the dimension increases the distances from a query point to the nearest and to the furthest neighbor converge. This implies that all points are concentrated close to a sphere around the query point, which makes it hard for an algorithm to discriminate the nearest neighbor. These are called *unstable* queries. In these cases a linear scan is again better than any known indexing method. The authors show that clustered data is *stable*, if the query point does not fall within a cluster; otherwise, the query is again unstable.

These results suggest that the assumptions commonly used for performance evaluation are not appropriate for discriminating between indexes. Empirical evidence indicates that the uniform distribution is the hardest case for the indexes, and at the same time unrealistic. Ciaccia et al. [25] propose the *homogeneity* assumption: given any two points in space (which we call the *viewpoints*) the distances from the viewpoints to the points in the database are almost identically distributed. The authors claim that real data follow this assumption. They support their claim by showing that their estimates are validated by experimental data.

Faloutsos and Kamel [40] take a different approach: they suggest that real data have properties of mathematical fractals, self similarity in particular. They propose the fractal dimension as a good tool for determining the performance of queries over data sets. Let P be a set of n points, and let $N(r)$ be the number of hyper-cubic grid cells of side r that contain at least one point from P. For self-similar data the log-log plot of $N(r)$ over r is almost a straight line. The slope of that line is the fractal dimension of the dataset P. Formally, for datasets that exhibit the self similarity property, the fractal dimension

f is defined as

$$f = -\frac{\partial \log N(r)}{\partial \log r} = \text{const.}$$

Therefore $N(r) = n/r^f$, which gives a good characterization of the distribution of the points in P.

The authors prove that there exist real datasets that exhibit fractal properties. They use the above observations to study the performance of indexes for range queries. It would be interesting to examine if this technique can be used for the evaluation of nearest neighbor queries.

Finally, an interesting theoretical model for the study of indexes is the *external memory* model introduced by Aggarwal and Vitter [2]. The model is characterized by the following parameters:

- N : the number of records of the problem instance,
- M : the number of records that can fit in the main memory,
- B : the number of records per block,

where $B > 1$, and $B \leq M < N$. Aggarwal and Vitter define an *I/O operation* in the model to be a swap of B records from the internal memory with B consecutive records from the external memory. The measure of the performance is the number of I/O operations needed to solve a problem. All other computation is free. They use this model to prove a $\Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ bound for the sorting problem. They also derive bounds for various geometrical problems [83]. It would be interesting to investigate the performance of nearest neighbor queries under this model.

8 Lower Bounds for the Nearest Neighbor Search Problem

To date there exists no satisfactory solution for the exact NNS problem either in theory, or in practice. It is widely believed among researchers that there exists no solution that achieves both poly(dn) storage space, and poly$(d \log n)$ query time for the exact problem. On the other hand, it has been shown that if we relax the problem to the approximate nearest neighbor search, then there are algorithms that meet the above requirements [61, 56], even though the storage space depends exponentially on $\mathcal{O}(\frac{1}{\varepsilon^2})$. One of the most interesting open problems in the field is to find a model in which we could establish the hardness of the NNS problem, and formally separate the exact from the approximate case. In this section, we examine some of the models that have can be utilized along this direction.

8.1 The cell probe model

One of the strongest theoretical models for the study of data structures is the *cell probe model*, introduced by Yao [88]. In this model, the data structure consists of a set of m cells, each one holding b bits of information. The query time for a search algorithm for the data structure is the number t of cells that are probed in order to answer the query.

A useful tool for the study of problems within the cell probe model is *asymmetric communication complexity* [71]. Asymmetric communication complexity is a special case of the communication complexity model, an elegant model introduced by Yao for the study of distributed applications. We have two players, Alice and Bob, that receive inputs x and y respectively, and wish to compute some function $f(x, y)$. The two players exchange bits of information according to some fixed protocol P. The cost of the protocol P on input (x, y) is the total number of bits exchanged between Alice and Bob. The cost of P is the maximum cost over all inputs. The complexity of the function f is the minimum cost of any protocol that computes f. In asymmetric communication complexity we distinguish between the bits sent by each player when we compute the cost of a protocol. In particular, we define a $[a, b]$-protocol for the function f as a protocol where on every input (x, y), Alice sends a total of at most a bits, and Bob sends a total of at most b bits. For data structure problems, we can think of the input x for Alice as being a query drawn from the set of all possible queries, while the input y for Bob is a data structure, drawn from the set of all possible data structures with n elements. The function $f(x, y)$ is the answer of the query x to the database y.

It was recently shown that there is a strong connection between asymmetric communication complexity, and the cell probe model. Miltersen [67, 68] proves the following theorem.

Theorem 1 *If there is a cell probe model solution for some problem that uses a data structure with m cells, b bits per cell, and answers queries with at most t probes, then there exists a deterministic asymmetric communication complexity protocol for this problem with $2t$ rounds of communication, where Alice sends at most $\log m$ bits in each of her messages, and Bob sends at most b bits in each of his messages*

Unfortunately, the converse can be proven only for the restricted case when the number of rounds of the protocol is constant.

Borodin et al. [15] study the *No Partial Match*(NPM) decision problem within the framework of the cell probe model. Let D be a database of n vectors in the d-dimensional Hamming space, and let q be a d-dimensional vector from the space $\{0, 1, *\}^d$. The query q matches some vector v in D, if for every coordinate $j \in \{1, 2, ..., d\}$ either $q_j = v_j$ or $q_j = *$. We refer to the coordinates of the query vector that are set to $*$, as the *don't care* coordinates, while the rest of the coordinates are called *exposed*. The NPM(D,q) decision problem returns YES if there is no vector in D that matches the query vector q, and returns NO otherwise.

Borodin et al. consider the NPM problem for the case that the query vector has exactly $\log n + 1$ exposed coordinates. Using the *richness* technique developed by Miltersen et al. [69], they prove that if there is an $[a, b]$-protocol for the NPM problem, then either $a = \Omega(\log n \log d)$, or $b = \Omega(n^{1-\varepsilon})$ for some $\varepsilon > 0$. Theorem 1 implies that any cell probe model algorithm for the exact partial match problem that makes t probes, either uses $2^{\Omega(\frac{1}{t}\log n \log d)}$ cells, or uses cells of size $\Omega(n^{1-\varepsilon}/t)$. At the extremes this implies the following:

- If the cell size is poly(d), and the number of cells is poly(n), then the algorithm must make $\Omega(\log d)$ probes. This improves upon the $\Omega(\sqrt{\log d})$ lower bound by Miltersen et al. [69].
- If the algorithm answers a query in a constant number of probes, then it either uses $2^{\Omega(\log n \log d)}$ cells, or it requires processing of a cell with $\Omega(n^{1-\varepsilon})$ size.

The authors prove however that the richness technique is quite unlikely to derive better lower bounds, or lower bounds for queries with more exposed bits. A better lower bound would imply a lower bound for branching programs, a notoriously difficult problem.

The NPM problem is of special importance for the NNS problem. Borodin et al. give a reduction from the NPM problem with λ "don't cares", to the problem λ-neighbor decision problem: given q query point q we return YES if there is a point within radius λ of the query point q, and NO otherwise. Therefore, lower bounds for the NPM problem imply lower bounds for the NNS problem. The above theorem suggests that any algorithm for the NNS problem that uses storage space polynomial in n and d, has query time $\Omega(\log d)$. Furthermore, if the algorithm can be modeled as a two-way communication protocol with constant number of rounds of communication, then either it uses space $\Omega(n^{\log d})$, or it has query time $\Omega(n^{1-\varepsilon})$, for some $\varepsilon > 0$.

In the same context, Chakrabarti et al. [21] provide lower bounds for deterministic algorithms for the problem of $(1+\varepsilon)$-approximate nearest neighbor. They show that if we construct a data structure that has poly(nd) number of cells, where each cell has size $\mathcal{O}(d)$, then the algorithm requires $\Omega(\log\log d/\log\log\log d)$ probes, for any $\varepsilon \leq 2^{\lceil \log^{1-\delta} n \rceil}$, for some $\delta > 0$. the Kushilevitz et al. algorithm can be implemented as a randomized protocol with polynomial number of cells, each containing d bits, that uses $\mathcal{O}(\log\log d)$ probes. However, the algorithm is randomized, so the two bounds are incomparable. It remains an interesting open problem to extend this lower bound for the case that we allow randomization.

The authors consider the case of the d-dimensional Hamming cube. Alice draws from the set Q of all possible queries, and Bob from the set P of all possible sets of points of size n. Let $P \times Q$ denote the set of all possible problem instances. At each round i some information is exchanged that restricts this set to the set of *admissible* problem instances $P_i \times Q_i$ that is consistent with the information exchanged so far. The authors construct a case such that after $t = \Theta(\log\log d/\log\log\log d)$ rounds the set $P_t \times Q_t$ contains two problem instances that have distinct answers, meaning that no point can serve as an approximate nearest neighbor for both instances.

8.2 The semi-group model

A well structured model for the study of lower bounds for search problems is Fredman's semi-group model [20, 45, 44]. A semi-group S is defined as a set closed under an associative addition operation.

(Sometimes, we may assume that the operation is also commutative). In this model, a semi-group value is associated with each data point. Let $U = \{v_1, v_2, ..., v_n\}$ be the set of semi-group values defined for the point in P. Let $Q_1, Q_2, ..., Q_m$ be subsets of the set P. A query is defined as the sum of the semi-group values associated with a set of points Q_j. A number of problems can be formulated in this way: for example, for decision problems, we make S the set of Boolean values under the OR operation; for reporting problems, we make S to be the set of points under the union operation.

The model allows one to precompute sums of arbitrary subsets of data points, and store them into a set of *registers*. The cost of a query is defined as the number of registers that must be summed in order to answer the query. The model can be applied to both *static*, and *semi-dynamic* problems. The two problems are actually equivalent. In the former, the semi-group values of the points are fixed, and we examine the tradeoff between query time and *storage redundancy*. In the latter, we are allowed to update the semi-group values of the points, and we are interested in the tradeoff between the query time and the update cost. Both the update cost and the storage redundancy for some variable v_i are defined as the total number of registers that contain this variable (by itself, or in a sum). Let U_i be the set of registers that hold the variable v_i. We define the *average storage redundancy* to be

$$\rho = \frac{\sum_{i=1}^{n} |U_i|}{n}.$$

Similarly, let R_j denote the set of registers that answer the query Q_j. We define the average query time to be

$$t = \frac{\sum_{i=1}^{m} |R_i|}{m}.$$

In [20] the authors provide a general technique for proving tradeoffs between the average storage redundancy, and the average query time. The idea is to construct a bipartite graph $G(V, E)$, called the *semantic graph*, where $V = U \cup R$, and $E = \{(i, j) \ : \ v_i \in Q_j\}$. We also assign a positive weight w_e to each edge $e \in E$. Let now B be a complete bipartite subgraph of G, and let U_B be the set of vertices of B from U, and R_B the set of vertices of B from R. Let u_B denote the size of the set U_B, and r_B the size of the set R_B. The authors [20, 45] prove that if we can find constants w_e, w_ρ, w_t such that for every complete bipartite subgraph B of G, we have that

$$\sum_{e \in E(B)} w_e \leq w_\rho \cdot u_B + w_t \cdot r_B, \tag{1}$$

then it holds that

$$\sum_{e \in E} w_e \leq w_\rho \cdot n \cdot \rho + w_t \cdot m \cdot t. \tag{2}$$

A rough intuition behind this lemma can be found by setting the weights w_e to one. In this case the sum $\sum_{e \in E} w_e$ represents the total cost of all possible queries in terms of accessed variables (not registers). Every edge that appears in G must be accounted for either in the cost of answering a query, or in the cost of including the variable in a register. The maximum between these two upper-bounds the total cost. This intuition becomes more clear if we make the set U_B to be the set of variables that appear in the register i, and the set R_B to be the set of queries that are answered by the register i. If we set $w_e = 1$ then $\sum_{e \in E(B)} w_e = u_B \cdot r_B$. The equation 1 says that this product is upper-bounded by either $w_\rho \cdot u_B$, or by $w_t \cdot r_B$. In particular, we select the constant w_ρ so that it represents the maximum possible value of r_B, that is, the maximum possible number of queries that are answered by a specific register. Therefore, we derive $w_\rho \cdot u_B$ as an upper bound on the cost (in terms of variables) of the queries that are answered by the register i, by overestimating the number of queries answered per register. Similarly, we select the constant w_t so that it represents the maximum possible value of u_B, that is, the maximum possible number of variables that appear in a specific register. Therefore, $w_t \cdot r_B$ is also an upper bound on the cost (in terms of variables) of the queries that are answered by the register i. In this case, we overestimate the number of variables per register. Summing over all possible registers, we obtain the equation 2.

Using this general technique the authors prove lower bounds for a variety of problems. Fredman [43] considers range queries, and spherical queries in the dynamic setting [44]. Fredman and Volper [46] consider the problem of partial match queries. They prove that the *complexity* of the problem is $1.226^d N$, where the complexity of a query is defined as the total number of registers accessed by a sequence of N update and query operations. This implies that there exists at least one query, or update operation that has cost 1.226^d. Therefore for any algorithm for the partial match problem, the total storage space is $1.226^d n$, or the query time is 1.226^d. This result relies on a more refined result on r-rank queries. The rank of a partial match query is the number of coordinates set to "*". They prove that for any ℓ, such that $0 \leq \ell \leq r$, if the algorithm for the r-rank queries has storage redundancy $\rho_r \leq \frac{1}{2}\binom{d}{\ell}/\binom{r}{\ell}$, then $t_r > 2^{r-\ell-1}$. The intuition behind the number ℓ is that the algorithm precomputes and stores the results of ℓ-rank queries, which it uses to answer r-rank queries.

The semi-group model derives stronger lower bounds but it is not as general as the cell probe model, because of the restrictions it imposes on the way that the data can be represented and manipulated. It is not clear for example that this model can be used for the representation of hashing schemes.

8.3 The bucketing model

A model oriented towards the study of hashing schemes is the model introduced by Dolev et al. [35, 34], which we shall refer to as the *bucketing model*. Let D be a set of words (dictionary) of length d over an alphabet Σ. The authors consider algorithms that map the dictionary D onto a set of m buckets $\{B_1, B_2, ..., B_m\}$, such that each word is mapped onto one or more buckets. We consider mappings that do not depend on the dictionary D.

Given a query point q, we are interested in the λ-neighborhood problem: given a query point q

return all points that are within distance λ from q. Let S_q denote the set of buckets accessed by the algorithm in order to answer the query. The time complexity of the query is defined as $TIME(q, P) = |S_q| + \sum_{i \in S_q} |B_i|$. Let $N_\lambda(q, D)$ denote the λ-neighborhood of the query point q with respect to D. A time optimal algorithm takes time $1 + |N_\lambda(q, D)|$ to answer the query. The time complexity of an algorithm A is defined as

$$T_A = \max_{q,D} \frac{TIME(q, D)}{1 + |N_\lambda(q, D)|}.$$

The space complexity of an algorithm A is defined as

$$S_A = \max_{D} \frac{\sum_{i=1}^{m} |B_i|}{|D|}.$$

A time optimal algorithm is one that for every possible query point q stores $|N_\lambda(q, D)|$ in a separate bucket. In the worst case, the space complexity of this algorithm is N_λ, where N_λ denotes the set of points in Σ^d that are contained in a ball of radius λ. A space optimal algorithm stores each point in a separate bucket. The time complexity of this algorithm is N_λ, since it must access all buckets even if they are empty.

Dolev et al. [35] prove formally this inherent tradeoff. The underlying idea behind their proof is that if we the algorithm has small space requirements, then many queries access the same bucket; therefore, the bucket contains many irrelevant words, resulting in large query time. In the extreme cases the authors show that if $T_A = \mathcal{O}(1)$ then $S_A = \Omega(N_\lambda)$, and if $S_A = \mathcal{O}(1)$ then $T_A = \Omega(\sqrt{N_\lambda})$. The last lower bound is improved to $\Omega(N_\lambda/4^\lambda)$ in [34]. The proof uses the idea of (λ, k)- coloring: we try to color the vertices of the cube Σ^d, such that the λ-neighborhood of any point is colored using at most k colors. Each color corresponds to a bucket. The bound on the number of colors gives a bound on the size of the sequence S_q. The size of each color class corresponds to the size of the bucket. The authors prove that if k is small then there exists some large color class, i.e. there exists a bucket with large size.

The authors also give upper bounds for the $(1, k)$-coloring problem in some special cases, using ideas from the area of error correcting codes. An (d, t, δ)-code is a collection of words of length d, such that the spheres of radius t around the code words are disjoint, and the spheres of radius $t + \delta$ cover the cube Σ^d. The algorithms for the λ-neighborhood problem create a bucket (or color class) for each code word, and map each point to the bucket of the nearest code word. Error-correcting codes seem like an interesting idea to be explored further.

8.4 The indexing model

A model of similar flavor was introduced by Hellerstein et al. [52] for the study of more general indexing schemes. Indexes in this model are evaluated in the context of a specific *workload*. A workload is defined by a triplet $W = (D, I, \mathcal{Q})$, where D is the domain of the workload (e.g. the space $\mathbb{R}^d$), I is a finite subset of the domain of size n, called an *instance*, and $\mathcal{Q}$ is a set of subsets of I, that represents the set of allowable queries over I. An *indexing scheme* $\mathcal{S} = \{S_1, ..., S_s\}$ defines an allocation (with possible replications) of the elements of I into s blocks of fixed size. The block size B is a fixed parameter of the system.

We are interested in studying tradeoffs between storage redundancy and access cost, for various workloads. For some indexing scheme $\mathcal{S}$ for some workload W, and for some element $x \in I$, we define the storage redundancy $\rho(x)$ of the element x, to be the total number of blocks in $\mathcal{S}$ that contain the element x. The *worst-case storage redundancy* of the indexing scheme $\mathcal{S}$ is defined as the maximum $\rho(x)$ over all elements x. The *average storage redundancy* is defined as the average number of blocks that contain an element of I, that is, $\rho = \frac{sB}{n}$. The average storage redundancy can be thought of as the ratio between the total number of blocks used by the index scheme $\mathcal{S}$ over the minimum possible number of blocks that contain the elements in I.

We now define the access cost of a query $Q \in \mathcal{Q}$ as the minimum number of blocks in $\mathcal{S}$ that contain the elements in Q. We define the *access overhead* $\alpha(Q)$ of a query $Q \in \mathcal{Q}$ as the ratio between the access cost of Q over $\lceil |Q|/B \rceil$, where $\lceil |Q|/B \rceil$ is the minimum possible number of blocks that answer the query Q. The access overhead α of the indexing scheme $\mathcal{S}$ is defined as the maximum access overhead over all possible queries. Note that $1 \leq \alpha \leq B$.

Samoladas and Miranker [79] prove that for any workload W, if we fix the access overhead α, then if we can find a set of M queries that have large size (that is, greater than B), and small intersection (that is, $\mathcal{O}(\frac{B}{\alpha^2})$), then the storage redundancy is $\Omega(\frac{MB}{n})$. They apply this theorem for the case of d-dimensional range queries, and derive a lower bound $\Omega(\frac{\log B}{\log \alpha})$ for the storage redundancy. Range queries are also considered in [52, 60].

Another interesting family of workloads is derived by the *set inclusion queries*. For some $m > 1$, the domain D is the set of all possible subsets of the set $\{1, 2, ..., m\}$. The instance I is a subset of D. Given a set $S \in D$, we define a query as $Q_S = \{X \in I : X \subseteq S\}$, that is, all the sets in I that are subsets of the set S. The set $\mathcal{Q}$ is the set of all such Q_Ss. We call this the Set Inclusion problem. The authors show that the set inclusion queries are hard. For any redundancy ρ, and any access overhead α, $1 \leq \alpha \leq B$, we can find an instance I with size $n > \alpha B \rho$, such that any indexing scheme $\mathcal{S}$ has access overhead at least α.

Indyk [55] shows that the set inclusion problem can be reduced to the c-approximate 1-neighborhood problem for the real space under the L_∞ norm. Given a query point q, the c-approximate λ-neighborhood search returns all points within distance λ from q, and no points at distance greater than $c \cdot \lambda$ from q. We define a reduction from workload $(D_A, I_A, \mathcal{Q}_A)$ to a workload $(D_B, I_B, \mathcal{Q}_B)$ as functions $f : D_A \to D_B$, and $g : \mathcal{Q}_A \to \mathcal{Q}_B$, such that for any $Q \in \mathcal{Q}_A$, and $S \in D_A$, we have $S \in Q$ if and only if $f(S) \in g(Q)$. Let now $W = (D, I, \mathcal{Q})$ be a Set Inclusion problem, where $D = \{1, 2, ..., n\}$, and I is a subset of D of size n. Let α be the access overhead of W. Indyk shows that there is a reduction from the Set Inclusion problem to the c-approximate 1-neighborhood problem in a real space of dimension $\alpha \log n$.

The reduction has the property that if $S \in Q$ then $|f(S) - g(Q)|_\infty \leq \frac{1}{3}$, and $|f(S) - g(Q)|_\infty = 1$ otherwise. This implies that for any $1 \leq c \leq 3$, for any storage redundancy ρ, and for any access overhead $1 \leq \alpha \leq B$, there exists an instance of size $n > \alpha B \rho$ of the c-approximate 1-neighborhood problem in $\mathbb{R}^{\alpha \log n}$ under L_∞ norm, that has access overhead at least α.

9 Conclusions and Discussion

We surveyed some of the past and recent results for the problem of exact and approximate Nearest Neighbor Search. Starting from the low dimensional problem, we saw that there exist optimal algorithms for dimensions one and two. However, there exists no known optimal solution for dimension $d \geq 3$. The additional complexity seems to stem from the fact that the representation of the Voronoi diagram requires space $\Omega(n^2)$ for $d \geq 2$. Therefore, time optimal algorithms that utilize Voronoi diagrams, can not achieve optimal space. Space optimal algorithms [89] do not achieve optimal time. It is an interesting problem to understand what makes the 3-dimensional problem harder, and whether this is an inherent property of the problem, that is, there is no algorithm that achieves both optimal space and time in the 3 dimensions.

For arbitrary dimension d, the best known solution by Meiser [66] achieves time $\mathcal{O}(d^5 \log n)$, and requires space $\mathcal{O}(n^{2d})$. Space optimal algorithms require almost linear time [89]. Finding an algorithm that achieves an efficient solution for the exact NNS (that is, a solution with poly(nd) space, and poly$(d \log n)$ query time) would be a breakthrough result. However, it seems more likely that such an algorithm does not exist. The efforts of the research community have been directed towards proving lower bounds for the exact NNS problem, that would formally prove that there exists no solution that is efficient in both space and time. This is one of the most challenging open problems, but probably also one of the hardest. It seems more feasible to attempt to derive lower bounds for some restricted class of algorithms. A possible direction along these lines is the semi-group model by Fredman. It would be interesting to examine if the Fredman model can be combined with the formulation of the NNS problem as it is given by Yao and Yao [89].

Furthermore, there are spaces for which the exact NNS problem is not well understood. There exists no algorithm for the exact NNS problem in the Hamming space, apart from the trivial reduction to the real space. Note that a nearest neighbor search in the Hamming space can be performed in $\mathcal{O}(1)$ time, if we settle for $\mathcal{O}(d2^d)$ storage space. It is surprising that the Hamming space is so poorly understood, and it seems like a promising direction to examine further. It is possible that because of the discrete nature of the space, we can achieve a better solution than in the general case.

When the NNS problem is relaxed to its approximate form then we have seen that there are solutions that achieve poly$(d \log n)$ query time, with poly(nd) space. These solutions can be hardly characterized as efficient since the space depends exponentially on $\mathcal{O}(\varepsilon^{-2})$. It would be ideal if we could improve upon these results, although current techniques do not seem likely to produce better algorithms. The most interesting questions arise from the lower bound point of view: it would be interesting to understand the distinction between the approximate and the exact NNS problem. Furthermore, the $\Omega(\log\log d/\log\log\log d)$ lower bound for the approximate NNS problem by Chakrabarti et al. [21] applies only to deterministic algorithms. However, the algorithms that obtain the best results are all randomized. An important open problem is to extend the lower bound to include randomized algorithms.

The NNS problem has applications in many diverse areas. Depending on the specific characteristics of each case we can create different problems, by varying the distance metric of the problem. A distance metric commonly used in practice is the L_∞ norm. Indyk [55] shows that the $(1+\varepsilon)$-approximate NNS problem, for $\varepsilon = 4\log_{1+\rho}\log 4d + 3$, can be solved with $\mathcal{O}(dn^{1+\rho}\log n)$ storage, and query time $\mathcal{O}(d\log n)$, using techniques similar to the ones used for the L_1 and L_2 norm. It seems possible to improve upon this result, and match the upper bounds for the L_1 and L_2 norm.

Another distance metric that is defined for the case of strings, and it is widely used in biological applications is the *edit distance* (or *Leverstein distance*). Let Σ denote an alphabet. We define the following set of operations on the elements of Σ: `insert`(c), `delete`(c), and `swap`(c_1, c_2). Each operation is associated with a cost. The cost of the `swap` operation is defined as an $\Sigma \times \Sigma$ matrix. The cost of the `insert` and `delete` operations may be defined as a constant, or it may depend on the character they are applied to. We define the edit distance between two strings s_1 and s_2 as the cost of the least expensive sequence of operations that converts s_1 to s_2.

The only known algorithm for performing NNS under the edit distance, performs a brute force search, using a dynamic programming algorithm for computing the edit distance. If we assume that all strings have length $\mathcal{O}(d)$, the algorithm uses space $\mathcal{O}(nd)$, and has query time $\mathcal{O}(nd^2)$. An interesting variation of the problem is considered by Dolev et al. [35]. Given two strings, s and q they compute the substring of s that has smallest edit distance from q. This is known as the *local alignment* problem, and it is strongly related to the NNS problem. Their algorithm uses error correcting codes, a technique that seems promising to explore further. In the worst case however, it performs as poorly as a sequential scan.

Interesting distance metrics are defined in information retrieval algorithms. The similarity between two documents is usually defined as the dot product of the two vectors, or the cosine of their angle. Recently, an interesting similarity measure for documents used for web search is the *set similarity* measure [18, 17]. Given two sets of terms A and B, we define the measure of their similarity as $\frac{A\cap B}{A\cup B}$. Indyk and Motwani consider the NNS problem under this similarity measure using locality sensitive hashing techniques and they obtain an algorithm with poly(nd) space, and sublinear time.

Another intriguing problem is the partial match problem. Borodin et al. [15] showed that we can reduce the partial match problem to the NNS problem. This means that lower bounds for the partial match problem imply lower bounds for the NNS problem. Furthermore, we can combine partial match and NNS problems; given a query point with k exposed coordinates, find the database point that is closer to this point with respect to this coordinates.

A number of questions remain unresolved in the area of indexing for nearest neighbor queries. Even the the 2-dimensional case is not well understood. For multidimensional spaces, all existing indexes are shown to perform poorly both in the worst case, as well as for many commonly used distributions. Finding realistic input distributions, as well as giving guarantees for these distributions remains an interesting research problem; for most cases indexes are evaluated experimentally. On a more abstract level, we would like to give some guarantees for the problem of NNS for *any* index. A promising work in this direction seems to be the work of Hellerstein et al. [52], where they provide a model oriented directly to the study of indexing schemes. Indyk [55] gives a lower bound for the λ-neighborhood problem for the case of the infinity norm. It seems possible to extend these results for other norms, using the reduction described in [15] from partial match to NNS. However, the indexing model is probably too strong to model the NNS problem since it does not take into account any of the blocks accessed by the query when searching for the nearest neighbor, but instead considers only the the block that contains the actual answer. A more appropriate model for the study of nearest neighbor queries may be the external memory model [2], and it would be interesting to explore it further.

Finally, in this report we did not address the problem of dynamic NNS. There is an extensive literature on this subject especially for the exact problem. Furthermore, there seems to be a connection between the static and the dynamic problem; using the idea of persistence, solutions for the dynamic problem can be used for the static case.

References

[1] P. K. Agarwal and J. Matušek. Ray shooting and parametric search. In *Proceedings of the 24th Symposium on Theory of Computing*, pages 517–526, 1992.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[3] N. Alon and J. Spencer. *The Probabilistic Method.* Wiley, 1992.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[5] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, 1993.

[6] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, 1994.

[7] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[8] K. P. Bennett, U. Fayyad, and D. Geiger. Density-based indexing for nearest neighbor queries. Technical Report MSR-TR-98-58, Microsoft Research, 1998.

[9] S. Berchtold, C. Böhm, D. A. Keim, and H. P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *PODS '97. Proceedings of the Sixteenth ACM Symposium on Principles of Database Systems*, pages 78–86, 1997.

[10] S. Berchtold, C.Böhm, and D. A. Keim. High-dimensional index structures – improving the performance of multimedia databases. Submitted for publication, 1999.

[11] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree : An index structure for high-dimensional data. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, pages 28–39, 1996.

[12] M. Bern. Approximate closest-point queries in high dimensions. *Information Processing Letters*, 45(2):95–99, 1993.

[13] M. W. Berry, S. T. Dumais, and A. T. Shippy. A case study of the latent semantic indexing. Technical Report CS-95-271, U. T. Knoxville, Jan 1995.

[14] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "Nearest Neighbor" meaningfulΓ In *Proc. 7th Int. Conf. Data Theory, ICDT*, pages 217–235, 1999.

[15] A. Borodin, R. Ostrovsky, , and Y. Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In *Proceedings of the 31st ACM Symposium on Theory of Computing*, pages 312–321, 1999.

[16] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2, pages 357–368, New York, May 13–15 1997.

[17] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the Sixth International World Wide Web Conference*, pages 391–404, 1997.

[18] A. Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29, 1998.

[19] C. Buckley, A. Sighal, M. Mitra, and G. Salton. New retrieval approaches using smart: Trec 4. In National Institute of Standards and Technology, editors, *Proceedings of the Fourth Text Retrieval Conference*, 1995.

[20] W. A. Burkhard, M. L. Fredman, and D. J. Kleitman. Inherent complexity trade-offs for range query problems. *Theoretical Computer Science*, 16(3):279–290, December 1981.

[21] A. Chakrabarti, B. Chazelle, Benjamin Gum, and A. Lvov. A lower bound for the complexity of approximate nearest-neighbor searching on the hamming cube. In *Proceedings of the 31st ACM Symposium on Theory of Computing*, pages 305–311, 1999.

[22] T. M. Chan. Approximate nearest neighbor queries revisited. In *Proceedings of the 13th Annual Symposium on Computational Geometry*, pages 352–358, 1997.

[23] B. Chazelle. How to search in history. *Information and Control*, 64(1–3):77–99, 1985.

[24] B. Chazelle, M. Sharir, and E. Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. In *Proceedings of the 6th Annual Symposium on Computational Geometry (SCG '90)*, pages 23–33, 1990.

[25] P. Ciaccia, M. Patella, and P. Zezula. A cost model for similarity queries in metric spaces. In *PODS '98. Proceedings of the ACM Symposium on Principles of Database Systems*, pages 59–78, 1998.

[26] K. Clarkson. A randomized algorithm for closest-point queries. *SIAM Journal of Computing*, 17:830–847, 1988.

[27] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proceedings of the Tenth Annual Symposium on Computational Geometry*, pages 160–164, 1994.

[28] D. Comer. Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[29] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[30] S. Cost and S. Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–67, 1993.

[31] T. M. Cover and P. E. Heart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13:21–27, 1967.

[32] S. Deerwerster, S. T. Dumais, G. W. Furnas, T. K. Landaur, , and R. Harshman. Indexing by latent semantic analysis. *Journal of American Society for Information Science*, 41:391–407, 1990.

[33] D. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM Journal of Computing*, 5(2):181–186, 1976.

[34] D. Dolev, Y. Harari, N. Linial, N. Nisan, and M. Parnas. Neighborhood preserving hashing and approximate queries. In *Proceedings of the fifth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 251–259, 1994.

[35] D. Dolev, Y. Harari, and M. Parnas. Finding the neighbor of a query in a dictionary. In *Procceedings of the 2nd ISTCS*, pages 33–42, 1993.

[36] R. O. Duda and P. E. Heart. *Pattern Classification and Scene Analysis.* John Wiley and Sons, 1973.

[37] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.

[38] C. Faloutsos. *Searching Multimedia Databases by Content.* Kluwer Academic Publishers, 1996.

[39] C. Faloutsos, R. Barber, M. Flickner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying in image content. *Journal of Intelligent Information Systems*, 3:231–262, 1994.

[40] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of R- trees using the concept of fractal dimension. In *PODS '94. Proceedings of the Thirteenth ACM Symposium on Principles of Database Systems*, volume 13, pages 4–13, 1994.

[41] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the QBIC system. *IEEE Computer*, 28:23–32, 1995.

[42] P. Frankl and H. Maehara. The Johnson-Lindenstrauss lemma and the sphericity of some graphs. *Journal of Combinatorial Theory B*, 44:355–362, 1988.

[43] M. L. Fredman. A lower bound on the complexity of orthogonal range queries. *Journal of the ACM*, 28(4):696–705, October 1981.

[44] M. L. Fredman. Lower bounds on the complexity of some optimal data structures. *SIAM Journal on Computing*, 10(1):1–10, 1981.

[45] M. L. Fredman and D. J. Volper. Query time versus redundancy trade-offs for range queries. *Journal of Computer and System Sciences*, 23(3):355–365, 1981.

[46] M. L. Fredman and D. J. Volper. The complexity of partial match retrieval in a dynamic setting. *Journal of Algorithms*, 3(1):68–78, 1982.

[47] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. In *ACM Transactions on Mathematical Software*, volume 3, pages 209–226, 1977.

[48] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[49] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic, 1991.

[50] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, 1999. To appear.

[51] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 14, pages 47–57, 1984.

[52] J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *Proceedings of the Sixteenth ACM Symposium on Principles of Database Systems*, pages 249–256, 1997.

[53] G. R. Hjaltason and H. Samet. Ranking in spatial databases. *Lecture Notes in Computer Science*, 951:83–95, 1995.

[54] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 27:417–441, 1933.

[55] P. Indyk. On approximate nearest neighbors in non-euclidean spaces. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998.

[56] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[57] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 618–625, 1997.

[58] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2, pages 369–380, 1997.

[59] J. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 599–608, 1997.

[60] E. Koutsoupias and D. S. Taylor. Tight bounds for 2-dimensional indexing schemes. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 52–58, 1998.

[61] E. Kushilevitz, R. Ostrovsky, , and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the 30th ACM Symposium on Theory of Computing*, pages 614–623, 1998.

[62] N. Linial and O. Sasson. Non-expansive hashing. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 509–518, 1996.

[63] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[64] J. Matušek. Reporting points in half-spaces. In *Proceedings of the 35th Symposium on Foundations of Computer Science*, pages 207–215, 1991.

[65] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *Journal of ACM*, 30:852–865, 1983.

[66] S. Meiser. Point location in arrangements of hyperplanes. *Information and Computation*, 106(2):286–303, 1993.

[67] P. B. Miltersen. Lower bounds for Union-Split-Find related problems on random access machines. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing*, pages 625–634, 1994.

[68] P. B. Miltersen. On the cell probe complexity of polynomial evaluation. *Theoretical Computer Science*, 143(1):167–174, May 1995.

[69] P. B. Miltersen, N. Nisan, S. Safra, and A. Wigderson. On data structures and asymmetric communication complexity. *Journal of Computer and System Sciences*, 57(1):37–49, August 1998.

[70] K. Mulmuley. *Computational Geometry. An Introduction Through Randomized Algorithms.* Prentice Hall, 1994.

[71] N. Nisan and E. Kushilevitz. *Communication Complexity.* Cambridge, University Press, 1997.

[72] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparisons. *Proceedings of the National Academy of Science*, 85(8):2444–2448, 1988.

[73] A. Pentland, R. W. Picard, and S. Sclaroff. Photobook: Tools for content-based manipulation of image databases. In *Proceedings of the SPIE Conference on Storage and Retrieval of Image and Video Databases II*, 1994.

[74] P. Raghavan. Information retrieval algorithms: a survey. In *Eighth ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–18, 1997.

[75] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 71–79, 1995.

[76] G. Salton and M. J. McGill. *Introduction to modern Information Retrieval.* McGraw-Hill Book Company, 1983.

[77] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[78] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, Massachusetts, U.S.A., 1990.

[79] V. Samoladas and D. P. Miranker. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 44–51, 1998.

[80] N. Sarnak and R.E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 26:669–679, 1986.

[81] J. Snoeyink. Point location. In J. E. Goodman and J. O' Rourke, editors, *Handbook of Discrete and Computational Geometry.* CRC Press, 1997.

[82] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, November 1991.

[83] J. S. Vitter. External memory algorithms. In *PODS '98. Proceedings of the ACM Symposium on Principles of Database Systems, June, 1998, Seattle, Washington*, pages 119–128, 1998.

[84] M. S. Waterman. Efficient sequence alignment algorithms. *Journal of Technical Biology*, 108(3):333–337, 1984.

[85] R. Weber and S. Blott. An approximation-based data structure for similarity search. Technical Report 24, ESPRIT project HERMES (no. 1941), 1997.

[86] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of 24th International Conference on Very Large Data Bases, VLDB*, pages 194–205, 1998.

[87] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523, February 1996.

[88] A. C. Yao. Should tables be sortedΓ (extended abstract). In *19th Annual Symposium on Foundations of Computer Science*, pages 22–27, 1978.

[89] A. C. Yao and F. F. Yao. A general approach to d-dimension geometric queries. In *Proceedings of the 17th Symposium on Theory of Computing*, pages 163–168, 1985.

[90] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pages 311–321, 1993.

A Range Spaces and VC dimension

A *range space* S is defined as a pair $(U, \mathcal{R})$, where U is a universe of geometric objects, and $\mathcal{R}$ is a set of subsets of the universe U. The subsets R contained in $\mathcal{R}$ are called *ranges* and they can be thought of as regions of the universe U. For example if U is the set $\mathbb{R}^d$, we can define the range R to be the set of points that are contained in some open ball centered at some point in U. The set $\mathcal{R}$ is the set of all open balls centered in some point in U. Let X be a subset of U (e.g. the set of points in the database P). We will show that we can construct a random sample A of X with "good" properties.

We give the following definitions.

Definition 4 *Let $S = (U, \mathcal{R})$ be a range space, $X \subset U$, and $\varepsilon > 0$. The set $A \subset X$ is an ε-sample for X if for every range $R \in \mathcal{R}$,*

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

Definition 5 *Let $S = (U, \mathcal{R})$ be a range space, $X \subset U$, and $\varepsilon > 0$. The set $A \subset X$ is an ε-net for X if for every range $R \in \mathcal{R}$, if $A \cap R$ is empty, then $|X \cap R| \leq \varepsilon |X|$.*

The first definition says that for every range R the points of A in R follow the same distribution as the points of X in R up to an additive constant ε. Intuitively, this means that the distribution of the objects in the sample is similar to the distribution of the objects in X.

The intuition behind the second definition is that if there exists a large concentration of points from X in some region of the universe U, then the sample A will contain some object from this region.

Therefore, if there is a range R in $\mathcal{R}$ that contains no point of the sample set A, then it should contain only a small number of points of the set X. This property of the ε-nets is used in random sampling algorithms so as to decompose a set X of objects into many sets of small size.

We now show that there are small ε-samples and ε-nets for certain range spaces, and that they can be found with random sampling. Let $S = (U, \mathcal{R})$ be a range space. and let X be a subset U. The set $P_{\mathcal{R}}(X) = \{R \cap X \ : \ R \in \mathcal{R}\}$ is the *projection* of $\mathcal{R}$ on X. If $P_{\mathcal{R}}(X)$ contains all the subsets of X, that is, if all subsets of X can be written as an intersection with some range R, then we say that X is *shattered* by the set $\mathcal{R}$. The *Vapnik-Chervonenkis dimension* (or *VC-dimension*) of the range space S, denoted by $VC(S)$, is the maximum cardinality of a shattered subset of U. If there are arbitrarily large shattered subsets of U then $VC(S) = \infty$. If the space $S = (U, \mathcal{R})$ has finite VC-dimension, then the range space with ranges that are Boolean combinations of at most k ranges from $\mathcal{R}$ has also finite dimension. In particular, if the VC-dimension of S is d then the VC-dimension of the new range space is $2dk \log(dk)$.

Consider, for example the range space $S = (U, \mathcal{R})$ we described above, where U is $\mathbb{R}^d$, and $\mathcal{R}$ is the set of open balls centered in some point in $\mathbb{R}^d$. It is not hard to see that the VC-dimension of the range space is $d+1$, since no set of $d+2$ points can be shattered to all its subsets with ranges from $\mathcal{R}$. Consider for example, a set X of points on a plane. First note that this set can be shattered only if the points are all vertices of a convex polygon. It is easy to see that any set of three points can be shattered to all of its subsets by the set of open balls. Let now X be a set of four points. It is obvious that there are two non-adjacent points which we can not obtain as an intersection of X with some open ball.

The following two theorems [3] show that there are small ε-samples, and ε-nets for range spaces with bounded VC-dimension. Furthermore, we can easily generate such sets by random sampling.

Theorem 2 *There is a positive constant c such that if $(U, \mathcal{R})$ is a range space of VC-dimension at most d, $X \subset U$ is a finite set, and $\varepsilon, \delta > 0$, then a random set A of cardinality s of X, where*

$$s \geq \frac{c}{\varepsilon^2}(d \log \frac{d}{\varepsilon} + \log \frac{1}{\delta})$$

is an ε-sample for X with probability at least $1 - \delta$.

Theorem 3 *Let $(U, \mathcal{R})$ be a range space of VC-dimension at most d, $X \subset U$ is a finite set, and $0 \leq \varepsilon, \delta \leq 1$. Let A be a set obtained by s random independent draws from X, where*

$$s \geq \max\{\frac{4}{\varepsilon} \log \frac{2}{\delta}, \frac{8d}{\varepsilon} \log \frac{8d}{\varepsilon}\}.$$

Then A is an ε-net for X with probability at least $1 - \delta$.

Note that if the set A is an ε-net then it is also an ε-sample. Furthermore, in both cases the size of A does not depend on the size of X.

B Parametric Search

Parametric search [65, 70] is a powerful technique that reduces search problems to decision problems. We present the technique through its application to the vertical ray shooting problem. Given a set of hyperplanes H that define an upper convex polytope, and a query point q, let o be a point (which may lie at infinity) from where a ray $\vec{oq}$ emanates. We want to identify the first hyperplane in H that is intersected by the ray $\vec{oq}$. Assume that there exists some algorithm A, such that given a point x it can answer the question: "does the segment $\overline{ox}$ intersect any of the hyperplanes in $H\Gamma$". This is called the *empty intersection problem*. The point x is the *parameter* of the algorithm A. The value of x for which the answer of the algorithm turns from NO to YES is called the *critical* value of the parameter, and it is denoted by x^*. We make the algorithm to output CRITICAL, when it is run on x^*.

Note that for any value α of x, we can tell if $\alpha > x^*$, $\alpha = x^*$, or $\alpha < x^*$, simply by looking at the output of the algorithm A on α. In general, we can compute the sign of a polynomial, simply by comparing x^* with the roots of the polynomial. The algorithm for computing x^* simulates A on x^*, without knowing x^*. At every stage of the algorithm we keep an interval I of candidate values for x. Each time a comparison is to be made between polynomials on x^* we invoke the algorithm A to locate x^* among the roots of the polynomial. Thus, we determine the sigh of the polynomial, and also identify an interval of roots (ρ_i, ρ_{i+1}) in which x^* belongs. If x^* is equal to some root, or if $I \cap (\rho_i, \rho_{i+1})$ is empty then we stop. Otherwise we set I to be $I \cap (\rho_i, \rho_{i+1})$ and we continue until some root becomes equal to x^*. If T_A is the time of the algorithm A then this procedure takes time $\mathcal{O}(T_A^2)$. We can improve this with parallelization. If T_P steps of the algorithm A can be done in parallel in P processors, then we can prove that the running time can be reduced to $\mathcal{O}(T_P T_A \log P)$.

C The Kushilevitz et al. algorithm

Kushilevitz et al. [61] solve the problem in the Euclidean space by reducing it to an instance of $(1+\varepsilon)$-approximate NNS in the Hamming space. Given a query point q, the idea is to project the vector $q-x$ on quantized random lines, for all x in P. Then, they use the property that the length of the vectors is preserved on average. Again, the underlying technique is randomized testing, since each projection can be viewed as a randomized test.

The search structure is constructed as follows. For every pair of points (a, b) in P we construct a search structure for the points in the closed ball $B(a, \ell)$, where $\ell = ||a-b||$. The points in $B(a, \ell)$ are mapped onto a Hamming cube of dimension DS as follows. Take a set of D random lines that pass though the origin. At each line place S breakpoints that divide the interval $[-L, L]$ into equal intervals. Let now p be a point in $B(x, \ell)$, and let p_j denote the projection of p on the j-th line. The point p is mapped to a DS-dimensional binary vector $\eta(p)$, such that the bit $\eta_{ji}(p)$ is set to one if the projection

p_j lies after the i-th breakpoint. The Hamming distance between two points $\eta(p)$ and $\eta(q)$ depends upon the number of breakpoints that are contained between the corresponding projections. Note that the number of breakpoints S that we place is an appropriately chosen constant, while the number L increases with the radius ℓ. Therefore, as ℓ increases the number the breakpoints become more sparse, and the Hamming distance decreases. In a way, the radius ℓ measures the precision with which we measure the distances.

The actual data structure consists of n^2 Hamming cubes of dimension DS, and a search structure for each cube. For each cube, we also keep a set of D vectors. The number D is chosen to be $\mathcal{O}(d^2 \log d)$. This value is determined by a VC-dimension argument, so as to guarantee that the distance $|p_j - q_j|$ follows a "nice" distribution with high probability. In this case, the scaled distance $\frac{\|p-q\|}{\ell}$ is preserved in the mapping.

Given the data structure, and in light of the above the discussion, the search algorithm operates as follows. Initially we start with two points a_0 and b_0, that are furthest apart. Let $\ell_0 = \|a_0 - b_0\|$. The ball $B(a_0, \ell_0)$ contains all points in the database. At round j we consider two points a_j and b_j, and we probe the Hamming cube for the points in $B(a_j, \ell_j)$, where $\ell_j = \|a_j - b_j\|$. Let a_{min} denote the actual nearest neighbor point to the query point q, and let ℓ_{min} denote the minimum distance. We consider the following cases.

- $\frac{\ell_{min}}{\ell_j} > \lambda_2$ for some constant $\lambda_2 = \mathcal{O}(1/\varepsilon)$. This means that the point q is far enough from the set of points in $B(a_j, \ell_j)$, so any point in $B(a_j, \ell_j)$ is an $(1+\varepsilon)$-approximate nearest neighbor, and we may as well return a_j.

- $\lambda_1 \le \frac{\ell_{min}}{\ell_j} \le \lambda_2$, for constant $\lambda_1 > 0$, and $\lambda_2 = \mathcal{O}(1/\varepsilon)$. It can be proven that in this case the Hamming distance gives a good approximation of the actual distance. Thus the nearest neighbor in the Hamming cube corresponds to the nearest neighbor in the $\mathbb{R}^d$ space.

- $\frac{\ell_{min}}{l_j} < \lambda_1$, for some constant $\lambda_1 > 0$. This means that the distance ℓ_j is not a good measure to discriminate between points that are close to each other, and we need to reduce it. We do so in a way that ensures that both the radius and the number of points to be considered reduces by a constant factor.

The algorithm finds an $(1+\varepsilon)$-approximate nearest neighbor in $\mathcal{O}(\log n)$ iterations. Each iteration performs $\mathcal{O}(\mathrm{poly}(\varepsilon^{-1}) d^2 \mathrm{poly}\log(nd/\varepsilon))$, and a search in the DS-dimensional Hamming cube that takes time $\mathcal{O}(\mathrm{poly}(\varepsilon^{-1}) d \mathrm{poly}\log(nd/\varepsilon))$. We can therefore conclude that the expected time of the algorithm is $\mathcal{O}(\mathrm{poly}(\varepsilon^{-1} d \log n)$. The space required for the data structure is $\mathcal{O}(\mathrm{poly}(\varepsilon^{-1})(n \log(dn/\varepsilon))^{O(\varepsilon^{-2})})$.