Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions

by Alexandr Andoni and Piotr Indyk

Abstract

In this article, we give an overview of efficient algorithms for the approximate and exact nearest neighbor problem. The goal is to preprocess a dataset of objects (e.g., images) so that later, given a new query object, one can quickly return the dataset object that is most similar to the query. The problem is of significant interest in a wide variety of areas.

The goal of this article is twofold. In the first part, we survey a family of nearest neighbor algorithms that are based on the concept of *locality-sensitive hashing*. Many of these algorithm have already been successfully applied in a variety of practical scenarios. In the second part of this article, we describe a recently discovered hashing-based algorithm, for the case where the objects are points in the d-dimensional Euclidean space. As it turns out, the performance of this algorithm is provably near-optimal in the class of the locality-sensitive hashing algorithms.

1 Introduction

The *nearest neighbor* problem is defined as follows: given a collection of n points, build a data structure which, given any query point, reports the data point that is closest to the query. A particularly interesting and well-studied instance is where the data points live in a d-dimensional space under some (e.g., Euclidean) distance function. This problem is of major importance in several areas; some examples are data compression, databases and data mining, information retrieval, image and video databases, machine learning, pattern recognition, statistics and data analysis. Typically, the features of each object of interest (document, image, etc.) are represented as a point in $\mathbb{R}^d$ and the distance metric is used to measure the similarity of objects. The basic problem then is to perform indexing or similarity searching for query objects. The number of features (i.e., the dimensionality) ranges anywhere from tens to millions. For example, one can represent a 1000×1000 image as a vector in a 1,000,000-dimensional space, one dimension per pixel.

There are several efficient algorithms known for the case when the dimension d is low (e.g., up to 10 or 20). The first such data structure, called *kd-trees* was introduced in 1975 by Jon Bentley [6], and remains one of the most popular data structures used for searching in multidimensional spaces. Many other multidimensional data structures are known, see [35] for an overview. However, despite decades of intensive effort, the current solutions suffer from either space or query time that is *exponential* in d. In fact, for large enough d, in theory or in practice, they often provide little improvement over a linear time algorithm that compares a query to each point from the database. This phenomenon is often called "the curse of dimensionality."

In recent years, several researchers have proposed methods for overcoming the running time bottleneck by using approximation (e.g., [5, 27, 25, 29, 22, 28, 17, 13, 32, 1], see also [36, 24]). In this formulation, the algorithm is allowed to return a point whose distance from the query is at most c times the distance from the query to its nearest points; $c > 1$ is called the *approximation factor*. The appeal of this approach is that, in many cases, an approximate nearest neighbor is almost as good as the exact one. In particular, if the distance measure accurately captures the notion of user quality, then small differences in the distance should not matter. Moreover, an efficient approximation algorithm can be used to solve the exact nearest neighbor problem by enumerating all approximate nearest neighbors and choosing the closest point[1].

In this article, we focus on one of the most popular algorithms for performing approximate search in high dimensions based on the concept of *locality-sensitive hashing* (LSH) [25]. The key idea is to hash the points using several hash functions to ensure that for each function the probability of collision is much higher for objects that are close to each other than for those that are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point.

The LSH algorithm and its variants has been successfully applied to computational problems in a variety of areas, including web clustering [23], computational biology [10.11], computer vision (see selected articles in [23]), computational drug design [18] and computational linguistics [34]. A code implementing a variant of this method is available from the authors [2]. For a more theoretically-oriented overview of this and related algorithms, see [24].

The purpose of this article is twofold. In Section 2, we describe the basic ideas behind the LSH algorithm and its analysis; we also give an overview of the current library of LSH functions for various distance measures in Section 3. Then, in Section 4, we describe a recently developed LSH family for the Euclidean distance, which achievies a near-optimal separation between the collision probabilities of close and far points. An interesting feature of this family is that it effectively enables the reduction of the approximate nearest neighbor problem for worst-case data to the exact nearest neighbor problem over random (or pseudorandom) point configuration in low-dimensional spaces.

Biographies

Alexandr Andoni (andoni@mit.edu) is a Ph.D. Candidate in computer science at Massachusetts Institute of Technology, Cambridge, MA.

Piotr Indyk (indyk@theory.lcs.mit.edu) is an associate professor in the Theory of Computation Group, Computer Science and Artificial Intelligence Lab, at Massachusetts Institute of Technology, Cambridge, MA.

[1] See section 2.4 for more information about exact algorithms.

Currently, the new family is mostly of theoretical interest. This is because the asymptotic improvement in the running time achieved via a better separation of collision probabilities makes a difference only for a relatively large number of input points. Nevertheless, it is quite likely that one can design better pseudorandom point configurations which do not suffer from this problem. Some evidence for this conjecture is presented in [3], where it is shown that point configurations induced by so-called *Leech lattice* compare favorably with truly random configurations.

Preliminaries

2.1 Geometric Normed Spaces

We start by introducing the basic notation used in this article. First, we use P to denote the set of data points and assume that P has cardinality n. The points p from P belong to a d-dimensional space $\mathbb{R}^d$. We use p_i to the denote the ith coordinate of p, for $i = 1 \ldots d$.

For any two points p and q, the distance between them is defined as

$$\|p - q\|_s = \left(\sum_{i=1}^{d} |p_i - q_i|^s \right)^{1/s}$$

for a parameter $s > 0$; this distance function is often called the ℓ_s *norm*. The typical cases include $s = 2$ (the Euclidean distance) or $s = 1$ (the Manhattan distance)[2]. To simplify notation, we often skip the subscript 2 when we refer to the Euclidean norm, that is, $\|p - q\| = \|p - q\|_2$.

Occasionally, we also use the Hamming distance, which is defined as the number of positions on which the points p and q differ.

2.2 Problem Definition

The nearest neighbor problem is an example of an *optimization* problem: the goal is to find a point which minimizes a certain objective function (in this case, the distance to the query point). In contrast, the algorithms that are presented in this article solve the decision version of the problem. To simplify the notation, we say that a point p is an *R-near neighbor* of a point q if the distance between p and q is at most R (see Figure 1). In this language, our algorithm either returns one of the R-near neighbors or concludes that no such point exists for some parameter R.

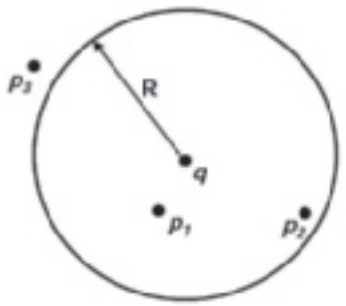


***Fig. 1. An illustration of an* R*-near neighbor query. The nearest neighbor of the query point* q *is the point* p_1*. However, both* p_1 *and* p_2 *are* R*-near neighbors of* q.

Naturally, the nearest and near neighbor problems are related. It is easy to see that the nearest neighbor problem also solves the R-near neighbor problem–one can simply check if the returned point is an R-near neighbor of the query point. The reduction in the other direction is somewhat more complicated and involves creating several instances of the near neighbor problem for different values of R. During the query time, the data structures are queried in the increasing order of R. The process is stopped when a data structure reports an answer. See [22] for a reduction of this type with theoretical guarantees.

In the rest of this article, we focus on the approximate near neighbor problem. The formal definition of the approximate version of the near neighbor problem is as follows.

Definition 2.1 (Randomized c*-approximate* R*-near neighbor, or* (c, R) – NN). Given a set P of points in a d-dimensional space $\mathbb{R}^d$, and parameters $R > 0$, $\delta > 0$, construct a data structure such that, given any query point q, if there exists an R-near neighbor of q in P, it reports some cR-near neighbor of q in P with probability $1 - \delta$.

For simplicity, we often skip the word randomized in the discussion. In these situations, we will assume that δ is an absolute constant bounded away from 1 (e.g., 1/2). Note that the probability of success can be amplified by building and querying several instances of the data structure. For example, constructing two independent data structures, each with $\delta = 1/2$, yields a data structure with a probability of failure $\delta = 1/2 \cdot 1/2 = 1/4$.

In addition, observe that we can typically assume that $R = 1$. Otherwise we can simply divide all coordinates by R. Therefore, we will often skip the parameter R as well and refer to the c-approximate near neighbor problem or c-NN.

We also define a related *reporting* problem.

Definition 2.2 (Randomized R*-near neighbor reporting).* Given a set P of points in a d-dimensional space $\mathbb{R}^d$, and parameters $R > 0$, $\delta > 0$, construct a data structure that, given any *query* point q, reports each R-near neighbor of q in P with probability $1 - \delta$.

Note that the latter definition does not involve an approximation factor. Also, unlike the case of the approximate near neighbor, here the data structure can return many (or even all) points if a large fraction of the data points are located close to the query point. As a result, one cannot give an a priori bound on the running time of the algorithm. However, as we point out later, the two problems are intimately related. In particular, the algorithms in this article can be easily modified to solve both c-NN and the reporting problems.

2.3 Locality-Sensitive Hashing

The LSH algorithm relies on the existence of *locality-sensitive hash functions*. Let $\mathcal{H}$ be a family of hash functions mapping $\mathbb{R}^d$ to some universe U. For any two points p and q, consider a process in which we choose a function h from $\mathcal{H}$ uniformly at random, and analyze the probability that $h(p) = h(q)$. The family $\mathcal{H}$ is called *locality sensitive* (with proper parameters) if it satisfies the following condition.

Definition 2.3 (Locality-sensitive hashing). A family $\mathcal{H}$ is called (R, cR, P_1, P_2)-sensitive if for any two points $p, q \in \mathbb{R}^d$.

- if $\|p - q\| \leq R$ then $\Pr_{\mathcal{H}}[h(q) = h(p)] \geq P_1$,
- if $\|p - q\| \geq cR$ then $\Pr_{\mathcal{H}}[h(q) = h(p)] \leq P_2$.

In order for a locality-sensitive hash (LSH) family to be useful, it has to satisfy $P_1 > P_2$.

[2] The name is motivated by the fact that $\|p - q\|_1 = \sum_{i=1}^{d} |p_i - q_i|$ is the length of the shortest path between p and q if one is allowed to move along only one coordinate at a time.

To illustrate the concept, consider the following example. Assume that the data points are *binary*, that is, each coordinate is either 0 or 1. In addition, assume that the distance between points p and q is computed according to the Hamming distance. In this case, we can use a particularly simple family of functions $\mathcal{H}$ which contains all projections of the input point on one of the coordinates, that is, $\mathcal{H}$ contains all functions h_i from $\{0, 1\}^d$ to $\{0, 1\}$ such that $h_i(p) = p_i$. Choosing one hash function h uniformly at random from $\mathcal{H}$ means that $h(p)$ returns a random coordinate of p (note, however, that different applications of h return the same coordinate of the argument).

To see that the family $\mathcal{H}$ is locality-sensitive with nontrivial parameters, observe that the probability $\Pr_{\mathcal{H}}[h(q) = h(p)]$ is equal to the fraction of coordinates on which p and q agree. Therefore, $P_1 = 1 - R/d$, while $P_2 = 1 - cR/d$. As long as the approximation factor c is greater than 1, we have $P_1 > P_2$.

2.4 The Algorithm

An LSH family $\mathcal{H}$ can be used to design an efficient algorithm for approximate near neighbor search. However, one typically cannot use $\mathcal{H}$ as is since the gap between the probabilities P_1 and P_2 could be quite small. Instead, an amplification process is needed in order to achieve the desired probabilities of collision. We describe this step next, and present the complete algorithm in the Figure 2.

Given a family $\mathcal{H}$ of hash functions with parameters (R, cR, P_1, P_2) as in Definition 2.3, we amplify the gap between the high probability P_1 and low probability P_2 by concatenating several functions. In particular, for parameters k and L (specified later), we choose L functions $g_j(q) = (h_{1,j}(q),\ldots,h_{k,j}(q))$, where $h_{t,j}$ $(1 \le t \le k,\ 1 \le j \le L)$ are chosen independently and uniformly at random from $\mathcal{H}$. These are the actual functions that we use to hash the data points.

The data structure is constructed by placing each point p from the input set into a bucket $g_j(p)$, for $j = 1,\ldots,L$. Since the total number of buckets may be large, we retain only the nonempty buckets by resorting to (standard) hashing[3] of the values $g_j(p)$. In this way, the data structure uses only $O(nL)$ memory cells; note that it suffices that the buckets store the pointers to data points, not the points themselves.

To process a query q, we scan through the buckets $g_1(q),\ldots, g_L(q)$, and retrieve the points stored in them. After retrieving the points, we compute their distances to the query point, and report any point that is a valid answer to the query. Two concrete scanning strategies are possible.

1. Interrupt the search after finding the first L' points (including duplicates) for some parameter L'.
2. Continue the search until all points from all buckets are retrieved; no additional parameter is required.

The two strategies lead to different behaviors of the algorithms. In particular, Strategy 1 solves the (c, R)-near neighbor problem, while Strategy 2 solves the R-near neighbor reporting problem.

Strategy 1. It is shown in [25, 19] that the first strategy, with $L' = 3L$, yields a solution to the *randomized c-approximate R-near neighbor problem*, with parameters R and δ for some constant failure probability $\delta < 1$. To obtain this guarantee, it suffices to set L to $\Theta(n^\rho)$, where $\rho = \frac{\ln 1/P_1}{\ln 1/P_2}$ [19]. Note that this implies that the algorithm runs in time proportional to n^ρ which is sublinear in n if $P_1 > P_2$. For example, if we use the hash functions for the binary vectors mentioned earlier, we obtain $\rho = 1/c$ [25, 19]. The exponents for other LSH families are given in Section 3.

Strategy 2. The second strategy enables us to solve the *randomized R-near neighbor reporting problem*. The value of the failure probability δ depends on the choice of the parameters k and L. Conversely, for each δ, one can provide parameters k and L so that the error probability is smaller than δ. The query time is also dependent on k and L. It could be as high as $\Theta(n)$ in the worst case, but, for many natural datasets, a proper choice of parameters results in a sublinear query time.

The details of the analysis are as follows. Let p be any R-neighbor of q, and consider any parameter k. For any function g_i, the probability that $g_i(p) = g_i(q)$ is at least P_1^k. Therefore, the probability that $g_i(p) = g_i(q)$ for *some* $i = 1\ldots L$ is at least $1 - (1 - P_1^k)^L$. If we set $L = \log_{1-P_1^k} \delta$ so that $(1 - P_1^k)^L \le \delta$, then any R-neighbor of q is returned by the algorithm with probability at least $1 - \delta$.

How should the parameter k be chosen? Intuitively, larger values of k lead to a larger gap between the probabilities of collision for close points and far points; the probabilities are P_1^k and P_2^k, respectively (see Figure 3 for an illustration). The benefit of this amplification is that the hash functions are more selective. At the same time, if k is large then P_1^k is small, which means that L must be sufficiently large to ensure that an R-near neighbor collides with the query point at least once.

[3] See [16] for more details on hashing.

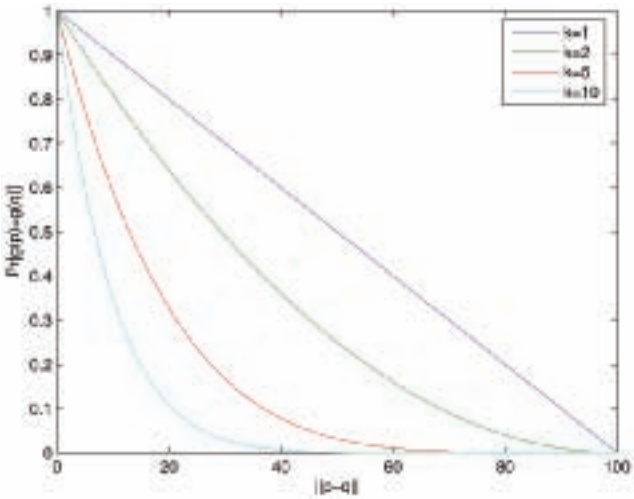
Preprocessing:

1. Choose L functions g_j, $j = 1,\ldots L$, by setting $g_j = (h_{1,j}, h_{2,j},\ldots h_{k,j})$, where $h_{1,j},\ldots h_{k,j}$ are chosen at random from the LSH family $\mathcal{H}$.
2. Construct L hash tables, where, for each $j = 1,\ldots L$, the j^{th} hash table contains the dataset points hashed using the function g_j.

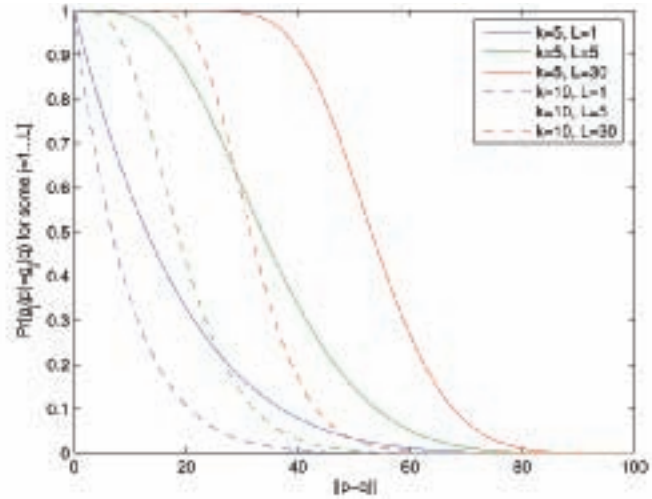
Query algorithm for a query point q:

1. For each $j = 1, 2,\ldots L$
   i) Retrieve the points from the bucket $g_j(q)$ in the j^{th} hash table.
   ii) For each of the retrieved point, compute the distance from q to it, and report the point if it is a correct answer (cR-near neighbor for Strategy 1, and R-near neighbor for Strategy 2).
   iii) (*optional*) Stop as soon as the number of reported points is more than L'.

Fig. 2. Preprocessing and query algorithms of the basic LSH algorithm.



(a) The probability that $g_j(p) = g_j(q)$ for a fixed j. Graphs are shown for several values of k. In particular, the blue function ($k = 1$) is the probability of collision of points p and q under a single random hash function h from the LSH family.



(b) The probability that $g_j(p) = g_j(q)$ for some $j = 1 \ldots L$. The probabilities are shown for two values of k and several values of L. Note that the slopes are sharper when k is higher.

Fig. 3. The graphs of the probability of collision of points p and q as a function of the distance between p and q for different values of k and L. The points p and q are d = 100 dimensional binary vectors under the Hamming distance. The LSH family $\mathcal{H}$ is the one described in Section 2.3.

A practical approach to choosing k was introduced in the E^{2}LSH package [2]. There the data structure optimized the parameter k as a function of the dataset and a set of sample queries. Specifically, given the dataset, a query point, and a fixed k, one can estimate precisely the expected number of collisions and thus the time for distance computations as well as the time to hash the query into all L hash tables. The sum of the estimates of these two terms is the estimate of the total query time for this particular query. E^{2}LSH chooses k that minimizes this sum over a small set of sample queries.

3 LSH Library

To date, several LSH families have been discovered. We briefly survey them in this section. For each family, we present the procedure of chosing a random function from the respective LSH family as well as its locality-sensitive properties.

Hamming distance. For binary vectors from $\{0, 1\}^d$, Indyk and Motwani [25] propose LSH function $h_i(p) = p_i$, where $i \in \{1, \ldots d\}$ is a randomly chosen index (the sample LSH family from Section 2.3). They prove that the exponent ρ is $1/c$ in this case.

It can be seen that this family applies directly to M-ary vectors (i.e., with coordinates in $\{1 \ldots M\}$) under the Hamming distance. Moreover, a simple reduction enables the extension of this family of functions to M-ary vectors under the ℓ_1 distance [30]. Consider any point p from $\{1 \ldots M\}^d$. The reduction proceeds by computing a binary string $Unary(p)$ obtained by replacing each coordinate p_i by a sequence of p_i ones followed by $M - p_i$ zeros. It is easy to see that for any two M-ary vectors p and q, the Hamming distance between $Unary(p)$ and $Unary(p)$ equals the ℓ_1 distance between p and q. Unfortunately, this reduction is efficient only if M is relatively small.

ℓ_1 distance. A more direct LSH family for $\mathbb{R}^d$ under the ℓ_1 distance is described in [4]. Fix a real $w \gg R$, and impose a randomly shifted grid with cells of width w; each cell defines a bucket. More specifically, pick random reals $s_1, s_2, \ldots s_d \in [0, w)$ and define $h_{s_1,\ldots,s_d} = (\lfloor (x_1 - s_1)/w \rfloor, \ldots, \lfloor (x_d - s_d)/w \rfloor)$. The resulting exponent is equal to $\rho = 1/c + O(R/w)$.

ℓ_s distance. For the Euclidean space, [17] propose the following LSH family. Pick a random projection of $\mathbb{R}^d$ onto a 1-dimensional line and chop the line into segments of length w, shifted by a random value $b \in [0, w)$. Formally, $h_{r,b} = \lfloor (r \cdot x + b)/w \rfloor$, where the projection vector $r \in \mathbb{R}^d$ is constructed by picking each coordinate of r from the Gaussian distribution. The exponent ρ drops strictly below $1/c$ for some (carefully chosen) finite value of w. This is the family used in the [2] package.

A generalization of this approach to ℓ_s norms for any $s \in [0, 2)$ is possible as well; this is done by picking the vector r from so-called *s-stable distribution*. Details can be found in [17].

Jaccard. To measure the similarity between two sets $A, B \subset U$ (containing, e.g., words from two documents), the authors of [9, 8] utilize the *Jaccard coefficient*. The Jaccard coefficient is defined as $s(A, B) = \frac{|A \cap B|}{|A \cup B|}$. Unlike the Hamming distance, Jaccard coefficient is a similarity measure: higher values of Jaccard coefficient indicate higher similarity of the sets. One can obtain the corresponding distance measure by taking $d(A, B) = 1 - s(A, B)$. For this measure, [9, 8] propose the following LSH family, called *min-hash*. Pick a random permutation π on the ground universe U. Then, define $h_\pi(A) = \min\{\pi(a) \mid a \in A\}$. It is not hard to prove that the probability of collision $\Pr_\pi[h_\pi(A) = h_\pi(B)] = s(A, B)$. See [7] for further theoretical developments related to such hash functions.

Arccos. For vectors $p, q \in \mathbb{R}^d$, consider the distance measure that is the angle between the two vectors, $\theta(p, q) = \arccos\left(\frac{p \cdot q}{\|p\| \cdot \|q\|}\right)$. For this distance measure, Charikar et al. (inspired by [20]) defines the following LSH family [14]. Pick a random unit-length vector $u \in \mathbb{R}^d$ and define $h_u(p) = \text{sign}(u \cdot p)$. The hash function can also be viewed as partitioning the space into two half-spaces by a randomly chosen hyperplane. Here, the probability of collision is $\Pr_u[h_u(p) = h_u(q)] = 1 - \theta(p, q)/\pi$.

ℓ_2 distance on a sphere. Terasawa and Tanaka [37] propose an LSH algorithm specifically designed for points that are on a unit hypersphere in the Euclidean space. The idea is to consider a regular polytope, orthoplex for example, inscribed into the hypersphere and rotated at random. The hash function then maps a point on the hypersphere into the closest polytope vertex lying on the hypersphere. Thus, the buckets of the hash function are the Voronoi cells of the polytope vertices lying on the hypersphere. [37] obtain exponent ρ that is an improvement over [17] and the Leech lattice approach of [3].

4 Near-Optimal LSH Functions for Euclidean Distance

In this section we present a new LSH family, yielding an algorithm with query time exponent $\rho(c) = 1/c^2 + O(\log \log n / \log^{1/3} n)$. For large enough n, the value of $\rho(c)$ tends to $1/c^2$. This significantly improves upon the earlier running time of [17]. In particular, for $c = 2$, our exponent tends to 0.25, while the exponent in [17] was around 0.45. Moreover, a recent paper [31] shows that hashing-based algorithms (as described in Section 2.3) cannot achieve $\rho < 0.462/c^2$. Thus, the running time exponent of our algorithm is essentially optimal, up to a constant factor.

We obtain our result by carefully designing a family of locality-sensitive hash functions in ℓ_2. The starting point of our construction is the line partitioning method of [17]. There, a point p was mapped into $\mathbb{R}^1$ using a random projection. Then, the line $\mathbb{R}^1$ was partitioned into intervals of length w, where w is a parameter. The hash function for p returned the index of the interval containing the projection of p.

An analysis in [17] showed that the query time exponent has an interesting dependence on the parameter w. If w tends to infinity, the exponent tends to $1/c$, which yields no improvement over [25, 19]. However, for small values of w, the exponent lies slightly below $1/c$. In fact, the unique minimum exists for each c.

In this article, we utilize a "multi-dimensional version" of the aforementioned approach. Specifically, we first perform random projection into $\mathbb{R}^t$, where t is super-constant, but relatively small (i.e., $t = o(\log n)$). Then we partition the space $\mathbb{R}^t$ into cells. The hash function function returns the index of the cell which contains projected point p.

The partitioning of the space $\mathbb{R}^t$ is somewhat more involved than its one-dimensional counterpart. First, observe that the natural idea of partitioning using a grid does not work. This is because this process roughly corresponds to hashing using concatenation of several one-dimensional functions (as in [17]). Since the LSH algorithms perform such concatenation anyway, grid partitioning does not result in any improvement. Instead, we use the method of "ball partitioning", introduced in [15], in the context of embeddings into tree metrics. The partitioning is obtained as follows. We create a sequence of balls $B_1, B_2 \ldots$, each of radius w, with centers chosen independently at random. Each ball B_i then defines a cell, containing points $B_i \setminus \cup_{j<i} B_j$.

In order to apply this method in our context, we need to take care of a few issues. First, locating a cell containing a given point could require enumeration of all balls, which would take an unbounded amount of time. Instead, we show that one can simulate this procedure by replacing each ball by a grid of balls. It is not difficult then to observe that a finite (albeit exponential in t) number U of such grids suffices to cover all points in $\mathbb{R}^t$. An example of such partitioning (for $t = 2$ and $U = 5$) is given in Figure 4.

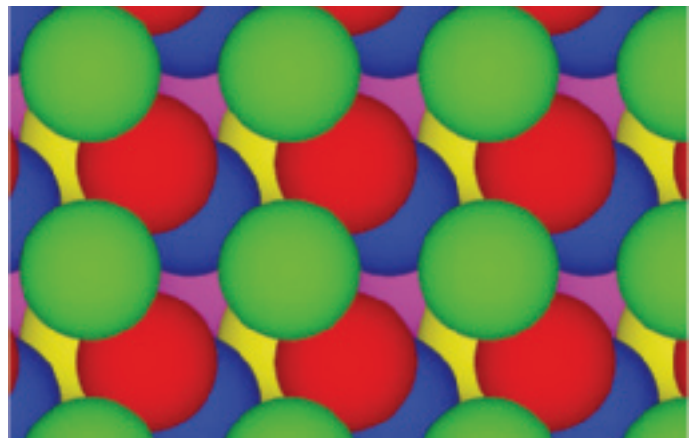


Fig. 4. An illustration of the the ball partitioning of the 2-dimensional space.

The second and the main issue is the choice of w. Again, it turns out that for large w, the method yields only the exponent of $1/c$. Specifically, it was shown in [15] that for any two points $p, q \in \mathbb{R}^t$, the probability that the partitioning separates p and q is at most $O(\sqrt{t} \cdot \|p - q\|/w)$. This formula can be showed to be tight for the range of w where it makes sense as a lower bound, that is, for $w = \Omega(\sqrt{t} \cdot \|p - q\|)$. However, as long as the separation probability depends linearly on the distance between p and q, the exponent ρ is still equal to $1/c$. Fortunately, a more careful analysis[4] shows that, as in the one-dimensional case, the minimum is achieved for finite w. For that value of w, the exponent tends to $1/c^2$ as t tends to infinity.

5 Related Work

In this section, we give a brief overview of prior work in the spirit of the algorithms considered in this article. We give only high-level simplified descriptions of the algorithms to avoid area-specific terminology. Some of the papers considered a closely related problem of finding all close pairs of points in a dataset. For simplicity, we translate them into the near neighbor framework since they can be solved by performing essentialy n separate near neighbor queries.

Hamming distance. Several papers investigated multi-index hashing-based algorithms for retrieving similar pairs of vectors with respect to the Hamming distance. Typically, the hash functions were projecting the vectors on some subset of the coordinates $\{1 \ldots d\}$ as in the example from an earlier section. In some papers [33, 21], the authors considered the probabilistic model where the data points are chosen uniformly at random, and the query point is a random point close to one of the points in the dataset. A different approach [26] is to assume that the dataset is arbitrary, but almost all points are far from the query point. Finally, the paper [12] proposed an algorithm which did not make any assumption on the input. The analysis of the algorithm was akin to the analysis sketched at the end of section 2.4: the parameters k and L were chosen to achieve desired level of sensitivity and accuracy.

Set intersection measure. To measure the similarity between two sets A and B, the authors of [9, 8] considered the Jaccard coefficient $s(A, B)$, proposing a family of hash functions $h(A)$ such that $\Pr[h(A) = h(B)] = s(A, B)$ (presented in detail in Section 3). Their main motivation was to

[4] Refer to [3] for more details.

construct short similarity-preserving "sketches" of sets, obtained by mapping each set A to a sequence $\langle h_1(A), ..., h_k(A)\rangle$. In section 5.3 of their paper, they briefly mention an algorithm similar to Strategy 2 described at the end of the Section 2.4. One of the differences is that, in their approach, the functions h_i are sampled without replacement, which made it more difficult to handle small sets.

Acknowledgement

This work was supported in part by NSF CAREER grant CCR-0133849 and David and Lucille Packard Fellowship.

References

1. Ailon, N. and Chazelle, B. 2006. Approximate nearest neighbors and the Fast Johnson-Lindenstrauss Transform. In *Proceedings of the Symposium on Theory of Computing*.
2. Andoni, A. and Indyk, P. 2004. E2lsh: Exact Euclidean locality-sensitive hashing. http://web.mit.edu/andoni/www/LSH/.
3. Andoni, A. and Indyk, P. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the Symposium on Foundations of Computer Science*.
4. Andoni, A. and Indyk, P. 2006. Efficient algorithms for substring near neighbor problem. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. 1203–1212.
5. Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., and Wu, A. 1994. An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. 573–582.
6. Bentley, J. L. 1975. Multidimensional binary search trees used for associative searching. *Comm. ACM 18*, 509–517.
7. Broder, A., Charikar, M., Frieze, A., and Mitzenmacher, M. 1998. Min-wise independent permutations. *J. Comput. Sys. Sci.*
8. Broder, A., Glassman, S., Manasse, M., and Zweig, G. 1997. Syntactic clustering of the web. In *Proceedings of the 6th International World Wide Web Conference*. 391–404.
9. Broder, A. 1997. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences*. 21–29.
10. Buhler, J. 2001. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinform. 17*, 419–428.
11. Buhler, J. and Tompa, M. 2001. Finding motifs using random projections. In *Proceedings of the Annual International Conference on Computational Molecular Biology* (*RECOMB1*).
12. Califano, A. and Rigoutsos, I. 1993. Flash: A fast look-up algorithm for string homology. In *Proceedings of the IEE Conference on Computer Vision and Pattern Recognition* (*CVPR*).
13. Chakrabarti, A. and Regev, O. 2004. An optimal randomised cell probe lower bounds for approximate nearest neighbor searching. In *Proceedings of the Symposium on Foundations of Computer Science*.
14. Charikar, M. 2002. Similarity estimation techniques from rounding. In *Proceedings of the Symposium on Theory of Computing*.
15. Charikar, M., Chekuri, C., Goel, A., Guha, S., and Plotkin, S. 1998. Approximating a finite metric by a small number of tree metrics. In *Proceedings of the Symposium on Foundations of Computer Science*.
16. Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. *Introduct. Algorithms*. 2nd Ed. MIT Press.
17. Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. 2004. Locality-sensitive hashing scheme based on p-stable distributions.In *Proceedings of the ACM Symposium on Computational Geometry*.
18. Dutta, D., Guha, R., Jurs, C., and Chen, T. 2006. Scalable partitioning and exploration of chemical spaces using geometric hashing. *J. Chem. Inf. Model. 46*.
19. Gionis, A., Indyk, P., and Motwani, R. 1999. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Databases*.
20. Goemans, M. and Williamson, D. 1995. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM 42*. 1115–1145.
21. Greene, D., Parnas, M., and Yao, F. 1994. Multi-index hashing for information retrieval. In Proceedings of the Symposium on Foundations of Computer Science. 722–731.
22. Har-Peled, S. 2001. A replacement for voronoi diagrams of near linear size. In *Proceedings of the Symposium on Foundations of Computer Science*.
23. Haveliwala, T., Gionis, A., and Indyk, P. 2000. Scalable techniques for clustering the web. *WebDB Workshop*.
24. Indyk, P. 2003. Nearest neighbors in high-dimensional spaces. In *Handbook of Discrete and Computational Geometry*. CRC Press.
25. Indyk, P. and Motwani, R. 1998. Approximate nearest neighbor: Towards removing the curse of dimensionality. In *Proceedings of the Symposium on Theory of Computing*.
26. Karp, R. M., Waarts, O., and Zweig, G. 1995. The bit vector intersection problem. In *Proceedings of the Symposium on Foundations of Computer Science*. pages 621–630.
27. Kleinberg, J. 1997. Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the Symposium on Theory of Computing*.
28. Krauthgamer, R. and Lee, J. R. 2004. Navigating nets: Simple algorithms for proximity search. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*.
29. Kushilevitz, E., Ostrovsky, R., and Rabani, Y. 1998. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the Symposium on Theory of Computing*. 614–623.
30. Linial, N., London, E., and Rabinovich, Y. 1994. The geometry of graphs and some of its algorithmic applications. In *Proceedings of the Symposium on Foundations of Computer Science*. 577–591.
31. Motwani, R., Naor, A., and Panigrahy, R. 2006. Lower bounds on locality sensitive hashing. In *Proceedings of the ACM Symposium on Computational Geometry*.
32. Panigrahy, R. 2006. Entropy-based nearest neighbor algorithm in high dimensions. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*.
33. Paturi, R., Rajasekaran, S., and Reif, J.The light bulb problem. *Inform. Comput. 117*, 2, 187–192.
34. Ravichandran, D., Pantel, P., and Hovy, E. 2005. Randomized algorithms and nlp: Using locality sensitive hash functions for high speed noun clustering. In *Proceedings of the Annual Meeting of the Association of Computational Linguistics*.
35. Samet, H. 2006. *Foundations of Multidimensional and Metric Data Structures*. Elsevier, 2006.
36. Shakhnarovich, G., Darrell, T., and Indyk, P. Eds. *Nearest Neighbor Methods in Learning and Vision*. Neural Processing Information Series, MIT Press.
37. Terasawa, T. and Tanaka, Y. 2007. Spherical lsh for approximate nearest neighbor search on unit hypersphere. In *Proceedings of the Workshop on Algorithms and Data Structures*.