Similarity Search in High Dimensions via Hashing

Aristides Gionis * Piotr Indyk† Rajeev Motwani‡
Department of Computer Science
Stanford University
Stanford, CA 94305
{gionis,indyk,rajeev}@cs.stanford.edu

Abstract

The nearest- or near-neighbor query problems arise in a large variety of database applications, usually in the context of similarity searching. Of late, there has been increasing interest in building search/index structures for performing similarity search over high-dimensional data, e.g., image databases, document collections, time-series databases, and genome databases. Unfortunately, all known techniques for solving this problem fall prey to the "curse of dimensionality." That is, the data structures scale poorly with data dimensionality; in fact, if the number of dimensions exceeds 10 to 20, searching in k-d trees and related structures involves the inspection of a large fraction of the database, thereby doing no better than brute-force linear search. It has been suggested that since the selection of features and the choice of a distance metric in typical applications is rather heuristic, determining an approximate nearest neighbor should suffice for most practical purposes. In this paper, we examine a novel scheme for approximate similarity search based on hashing. The basic idea is to hash the points from the database so as to ensure that the probability of collision is much higher for objects that are close to each other than for those that are far apart. We provide experimental evidence that our method gives significant improvement in running time over other methods for searching in high-dimensional spaces based on hierarchical tree decomposition. Experimental results also indicate that our scheme scales well even for a relatively large number of dimensions (more than 50).

*Supported by NAVY N00014-96-1-1221 grant and NSF Grant IIS-9811904.

†Supported by Stanford Graduate Fellowship and NSF NYI Award CCR-9357849.

‡Supported by ARO MURI Grant DAAH04-96-1-0007, NSF Grant IIS-9811904, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

A similarity search problem involves a collection of objects (e.g., documents, images) that are characterized by a collection of relevant features and represented as points in a high-dimensional attribute space; given queries in the form of points in this space, we are required to find the nearest (most similar) object to the query. The particularly interesting and well-studied case is the d-dimensional Euclidean space. The problem is of major importance to a variety of applications; some examples are: data compression [20]; databases and data mining [21]; information retrieval [11, 16, 38]; image and video databases [15, 17, 37, 42]; machine learning [7]; pattern recognition [9, 13]; and, statistics and data analysis [12, 27]. Typically, the features of the objects of interest are represented as points in $\Re^d$ and a distance metric is used to measure 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 thousands. For example, in multimedia applications such as IBM's QBIC (Query by Image Content), the number of features could be several hundreds [15, 17]. In information retrieval for text documents, vector-space representations involve several thousands of dimensions, and it is considered to be a dramatic improvement that dimension-reduction techniques, such as the Karhunen-Loéve transform [26, 30] (also known as principal components analysis [22] or latent semantic indexing [11]), can reduce the dimensionality to a mere few hundreds!

The low-dimensional case (say, for d equal to 2 or 3) is well-solved [14], so the main issue is that of dealing with a large number of dimensions, the so-called "curse of dimensionality." Despite decades of intensive effort, the current solutions are not entirely satisfactory; in fact, for large enough d, in theory or in practice, they provide little improvement over a linear algorithm which compares a query to each point from the database. In particular, it was shown in [45] that, both empirically and theoretically, *all* current indexing techniques (based on space partitioning) degrade to linear search for sufficiently high dimensions. This situation poses a serious obstacle to the future development of large scale similarity search systems. Imagine for example a search engine which enables content-based image retrieval on the World-Wide Web. If the system was to index a significant fraction of the web, the number of images to index would be at least of the order tens (if not hundreds) of million. Clearly, no indexing method exhibiting linear (or close to linear) dependence on the data size could manage such a huge data set.

The premise of this paper is that in many cases it is not necessary to insist on the *exact* answer; instead, determining an *approximate* answer should suffice (refer to Section 2 for a formal definition). This observation underlies a large body of recent research in databases, including using random sampling for histogram estimation [8] and median approximation [33], using wavelets for selectivity estimation [34] and approximate SVD [25]. We observe that there are many applications of nearest neighbor search where an approximate answer is good enough. For example, it often happens (e.g., see [23]) that the relevant answers are *much* closer to the query point than the irrelevant ones; in fact, this is a desirable property of a good similarity measure. In such cases, the approximate algorithm (with a suitable approximation factor) will return the same result as an exact algorithm. In other situations, an approximate algorithm provides the user with a time-quality tradeoff — the user can decide whether to spend more time waiting for the exact answer, or to be satisfied with a much quicker approximation (e.g., see [5]).

The above arguments rely on the assumption that approximate similarity search can be performed much faster than the exact one. In this paper we show that this is indeed the case. Specifically, we introduce a new indexing method for approximate nearest neighbor with a truly sublinear dependence on the data size even for high-dimensional data. Instead of using space partitioning, it relies on a new method called *locality-sensitive hashing (LSH)*. The key idea is to hash the points using several hash functions so as to ensure that, for each function, the probability of collision is much higher for objects which are close to each other than for those which are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point. We provide such locality-sensitive hash functions that are simple and easy to implement; they can also be naturally extended to the *dynamic* setting, i.e., when insertion and deletion operations also need to be supported. Although in this paper we are focused on Euclidean spaces, different LSH functions can be also used for other similarity measures, such as dot product [5].

Locality-Sensitive Hashing was introduced by Indyk and Motwani [24] for the purposes of devising main memory algorithms for nearest neighbor search; in particular, it enabled us to achieve worst-case $O(dn^{1/\epsilon})$-time for approximate nearest neighbor query over an n-point database. In this paper we improve that technique and achieve a significantly improved query time of $O(dn^{1/(1+\epsilon)})$. This yields an approximate nearest neighbor algorithm running in sublinear time for any $\epsilon > 0$. Furthermore, we generalize the algorithm and its analysis to the case of *external memory*.

We support our theoretical arguments by empirical evidence. We performed experiments on two data sets. The first contains 20,000 histograms of color images, where each histogram was represented as a point in d-dimensional space, for d up to 64. The second contains around 270,000 points representing texture information of blocks of large aerial photographs. All our tables were stored on disk. We compared the performance of our algorithm with the performance of the Sphere/Rectangle-tree (SR-tree) [28], a recent data structure which was shown to be comparable to or significantly more efficient than other tree-decomposition-based indexing methods for spatial data. The experiments show that our algorithm is significantly faster than the earlier methods, in some cases even by several orders of magnitude. It also scales well as the data size and dimensionality increase. Thus, it enables a new approach to high-performance similarity search — fast retrieval of approximate answer, possibly followed by a slower but more accurate computation in the few cases where the user is not satisfied with the approximate answer.

The rest of this paper is organized as follows. In Section 2 we introduce the notation and give formal definitions of the similarity search problems. Then in Section 3 we describe locality-sensitive hashing and show how to apply it to nearest neighbor search. In Section 4 we report the results of experiments with LSH. The related work is described in Section 5. Finally, in Section 6 we present conclusions and ideas for future research.

2 Preliminaries

We use l_p^d to denote the Euclidean space $\Re^d$ under the l_p norm, i.e., when the length of a vector $(x_1, \ldots x_d)$ is defined as $(|x_1|^p + \ldots + |x_d|^p)^{1/p}$. Further, $d_p(p, q) =$

$||p-q||_p$ denotes the distance between the points p and q in l_p^d. We use H^d to denote the *Hamming metric space* of dimension d, i.e., the space of binary vectors of length d under the standard Hamming metric. We use $d_H(p,q)$ denote the *Hamming distance*, i.e., the number of bits on which p and q differ.

The nearest neighbor search problem is defined as follows:

Definition 1 (Nearest Neighbor Search (NNS)) *Given a set P of n objects represented as points in a normed space l_p^d, preprocess P so as to efficiently answer queries by finding the point in P closest to a query point q.*

The definition generalizes naturally to the case where we want to return $K > 1$ points. Specifically, in the *K-Nearest Neighbors Search (K-NNS)*, we wish to return the K points in the database that are closest to the query point. The approximate version of the NNS problem is defined as follows:

Definition 2 (ϵ-Nearest Neighbor Search (ϵ-NNS)) *Given a set P of points in a normed space l_p^d, preprocess P so as to efficiently return a point $p \in P$ for any given query point q, such that $d(q,p) \leq (1+\epsilon)d(q,P)$, where $d(q,P)$ is the distance of q to the its closest point in P.*

Again, this definition generalizes naturally to finding $K > 1$ approximate nearest neighbors. In the *Approximate K-NNS* problem, we wish to find K points $p_1, \ldots, p_K$ such that the distance of p_i to the query q is at most $(1+\epsilon)$ times the distance from the ith nearest point to q.

3 The Algorithm

In this section we present efficient solutions to the approximate versions of the NNS problem. Without significant loss of generality, we will make the following two assumptions about the data:

1. the distance is defined by the l_1 norm (see comments below),
2. all coordinates of points in P are positive integers.

The first assumption is not very restrictive, as usually there is no clear advantage in, or even difference between, using l_2 or l_1 norm for similarity search. For example, the experiments done for the Webseek [43] project (see [40], chapter 4) show that comparing color histograms using l_1 and l_2 norms yields very similar results (l_1 is marginally better). Both our data sets (see Section 4) have a similar property. Specifically, we observed that a nearest neighbor of an average query point computed under the l_1 norm was also an ϵ-approximate neighbor under the l_2 norm with an average value of ϵ less than 3% (this observation holds for both data sets). Moreover, in most cases (i.e., for 67% of the queries in the first set and 73% in the second set) the nearest neighbors under l_1 and l_2 norms were *exactly* the same. This observation is interesting in its own right, and can be partially explained via the theorem by Figiel et al (see [19] and references therein). They showed analytically that by simply applying scaling and random rotation to the space l_2, we can make the distances induced by the l_1 and l_2 norms almost equal up to an arbitrarily small factor. It seems plausible that real data is already randomly rotated, thus the difference between l_1 and l_2 norm is very small. Moreover, for the data sets for which this property does not hold, we are guaranteed that after performing scaling and random rotation our algorithms can be used for the l_2 norm with arbitrarily small loss of precision.

As far as the second assumption is concerned, clearly all coordinates can be made positive by properly translating the origin of $\Re^d$. We can then convert all coordinates to integers by multiplying them by a suitably large number and rounding to the nearest integer. It can be easily verified that by choosing proper parameters, the error induced by rounding can be made arbitrarily small. Notice that after this operation the minimum interpoint distance is 1.

3.1 Locality-Sensitive Hashing

In this section we present locality-sensitive hashing (LSH). This technique was originally introduced by Indyk and Motwani [24] for the purposes of devising main memory algorithms for the ϵ-NNS problem. Here we give an improved version of their algorithm. The new algorithm is in many respects more natural than the earlier one: it does not require the hash buckets to store only one point; it has better running time guarantees; and, the analysis is generalized to the case of secondary memory.

Let C be the largest coordinate in all points in P. Then, as per [29], we can embed P into the Hamming cube $H^{d'}$ with $d' = Cd$, by transforming each point $p = (x_1, \ldots x_d)$ into a binary vector

$$v(p) = \text{Unary}_C(x_1) \ldots \text{Unary}_C(x_d),$$

where $\text{Unary}_C(x)$ denotes the unary representation of x, i.e., is a sequence of x ones followed by $C-x$ zeroes.

Fact 1 *For any pair of points p, q with coordinates in the set $\{1 \ldots C\}$,*

$$d_1(p,q) = d_H(v(p), v(q)).$$

That is, the embedding preserves the distances between the points. Therefore, in the sequel we can concentrate on solving ϵ-NNS in the Hamming space $H^{d'}$. However, we emphasize that we *do not* need to actually *convert* the data to the unary representation,

which could be expensive when C is large; in fact, all our algorithms can be made to run in time *independent* on C. Rather, the unary representation provides us with a convenient framework for description of the algorithms which would be more complicated otherwise.

We define the hash functions as follows. For an integer l to be specified later, choose l subsets $I_1, \ldots, I_l$ of $\{1, \ldots, d'\}$. Let $p_{|I}$ denote the projection of vector p on the coordinate set I, i.e., we compute $p_{|I}$ by selecting the coordinate positions as per I and concatenating the bits in those positions. Denote $g_j(p) = p_{|I_j}$. For the preprocessing, we store each $p \in P$ in the bucket $g_j(p)$, for $j = 1, \ldots, l$. As the total number of buckets may be large, we compress the buckets by resorting to standard hashing. Thus, we use two levels of hashing: the LSH function maps a point p to bucket $g_j(p)$, and a standard hash function maps the contents of these buckets into a hash table of size M. The maximal bucket size of the latter hash table is denoted by B. For the algorithm's analysis, we will assume hashing with chaining, i.e., when the number of points in a bucket exceeds B, a new bucket (also of size B) is allocated and linked to and from the old bucket. However, our implementation does not employ chaining, but relies on a simpler approach: if a bucket in a given index is full, a new point cannot be added to it, since it will be added to some other index with high probability. This saves us the overhead of maintaining the link structure.

The number n of points, the size M of the hash table, and the maximum bucket size B are related by the following equation:

$$M = \alpha \frac{n}{B},$$

where α is the memory utilization parameter, i.e., the ratio of the memory allocated for the index to the size of the data set.

To process a query q, we search all indices $g_1(q), \ldots, g_l(q)$ until we either encounter at least $c \cdot l$ points (for c specified later) or use all l indices. Clearly, the number of disk accesses is always upper bounded by the number of indices, which is equal to l. Let $p_1, \ldots, p_t$ be the points encountered in the process. For Approximate K-NNS, we output the K points p_i closest to q; in general, we may return fewer points if the number of points encountered is less than K.

It remains to specify the choice of the subsets I_j. For each $j \in \{1, \ldots, l\}$, the set I_j consists of k elements from $\{1, \ldots, d'\}$ sampled uniformly at random with replacement. The optimal value of k is chosen to maximize the probability that a point p "close" to q will fall into the same bucket as q, and also to minimize the probability that a point p' "far away" from q will fall into the same bucket. The choice of the values of l and k is deferred to the next section.

Algorithm Preprocessing
Input A set of points P,
  l (number of hash tables),
Output Hash tables $\mathcal{T}_i$, $i = 1, \ldots, l$
Foreach $i = 1, \ldots, l$
  Initialize hash table $\mathcal{T}_i$ by generating
  a random hash function $g_i(\cdot)$
Foreach $i = 1, \ldots, l$
  Foreach $j = 1, \ldots, n$
    Store point p_j on bucket $g_i(p_j)$ of hash table $\mathcal{T}_i$

Figure 1: Preprocessing algorithm for points already embedded in the Hamming cube.

Algorithm Approximate Nearest Neighbor Query
Input A query point q,
  K (number of appr. nearest neighbors)
Access To hash tables $\mathcal{T}_i$, $i = 1, \ldots, l$
  generated by the preprocessing algorithm
Output K (or less) appr. nearest neighbors
$S \leftarrow \emptyset$
Foreach $i = 1, \ldots, l$
  $S \leftarrow S \cup \{$points found in $g_i(q)$ bucket of table $\mathcal{T}_i\}$
Return the K nearest neighbors of q found in set S
/* Can be found by main memory linear search */

Figure 2: Approximate Nearest Neighbor query answering algorithm.

Although we are mainly interested in the I/O complexity of our scheme, it is worth pointing out that the hash functions can be efficiently computed if the data set is obtained by mapping l_1^d into d'-dimensional Hamming space. Let p be any point from the data set and let p' denote its image after the mapping. Let I be the set of coordinates and recall that we need to compute $p'_{|I}$. For $i = 1, \ldots, d$, let $I_{|i}$ denote, in sorted order, the coordinates in I which correspond to the ith coordinate of p. Observe, that projecting p' on $I_{|i}$ results in a sequence of bits which is monotone, i.e., consists of a number, say o_i, of ones followed by zeros. Therefore, in order to represent p'_I it is sufficient to compute o_i for $i = 1, \ldots, d$. However, the latter task is equivalent to finding the number of elements in the sorted array $I_{|i}$ which are smaller than a given value, i.e., the ith coordinate of p. This can be done via binary search in $\log C$ time, or even in constant time using a precomputed array of C bits. Thus, the total time needed to compute the function is either $O(d \log C)$ or $O(d)$, depending on resources used. In our experimental section, the value of C can be made very small, and therefore we will resort to the second method.

For quick reference we summarize the preprocessing

and query answering algorithms in Figures 1 and 2.

3.2 Analysis of Locality-Sensitive Hashing

The principle behind our method is that the probability of collision of two points p and q is closely related to the distance between them. Specifically, the larger the distance, the smaller the collision probability. This intuition is formalized as follows [24]. Let $D(\cdot,\cdot)$ be a distance function of elements from a set S, and for any $p \in S$ let $\mathcal{B}(p,r)$ denote the set of elements from S within the distance r from p.

Definition 3 *A family $\mathcal{H}$ of functions from S to U is called* (r_1, r_2, p_1, p_2)-sensitive *for $D(\cdot,\cdot)$ if for any $q, p \in S$*

- *if $p \in \mathcal{B}(q, r_1)$ then $\Pr_{\mathcal{H}}[h(q) = h(p)] \geq p_1$,*
- *if $p \notin \mathcal{B}(q, r_2)$ then $\Pr_{\mathcal{H}}[h(q) = h(p)] \leq p_2$.*

In the above definition, probabilities are considered with respect to the random choice of a function h from the family $\mathcal{H}$. In order for a locality-sensitive family to be useful, it has to satisfy the inequalities $p_1 > p_2$ and $r_1 < r_2$.

Observe that if $D(\cdot,\cdot)$ is the Hamming distance $d_H(\cdot,\cdot)$, then the family of projections on one coordinate is locality-sensitive. More specifically:

Fact 2 *Let S be $H^{d'}$ (the d'-dimensional Hamming cube) and $D(p,q) = d_H(p,q)$ for $p, q \in H^{d'}$. Then for any r, $\epsilon > 0$, the family $\mathcal{H}_{d'} = \{h_i : h_i((b_1, \ldots, b_{d'})) = b_i, \text{ for } i = 1, \ldots, d'\}$ is $\left(r, r(1+\epsilon), 1 - \frac{r}{d'}, 1 - \frac{r(1+\epsilon)}{d'}\right)$-sensitive.*

We now generalize the algorithm from the previous section to an *arbitrary* locality-sensitive family $\mathcal{H}$. Thus, the algorithm is equally applicable to other locality-sensitive hash functions (e.g., see [5]). The generalization is simple: the functions g are now defined to be of the form

$$g_i(p) = (h_{i_1}(p), h_{i_2}(p), \ldots, h_{i_k}(p)),$$

where the functions $h_{i_1}, \ldots, h_{i_k}$ are randomly chosen from $\mathcal{H}$ with replacement. As before, we choose l such functions $g_1, \ldots, g_l$. In the case when the family $\mathcal{H}_{d'}$ is used, i.e., each function selects one bit of an argument, the resulting values of $g_j(p)$ are essentially equivalent to $p_{|I_j}$.

We now show that the LSH algorithm can be used to solve what we call the (r,ϵ)-Neighbor problem: determine whether there exists a point p within a fixed distance $r_1 = r$ of q, or whether all points in the database are at least a distance $r_2 = r(1+\epsilon)$ away from q; in the first case, the algorithm is required to return a point p' within distance at most $(1+\epsilon)r$ from q. In particular, we argue that the LSH algorithm solves this problem for a proper choice of k and l, depending on r and ϵ. Then we show how to apply the solution to this problem to solve ϵ-NNS.

Denote by P' the set of all points $p' \in P$ such that $d(q,p') > r_2$. We observe that the algorithm correctly solves the (r,ϵ)-Neighbor problem if the following two properties hold:

P1 If there exists p^* such that $p^* \in \mathcal{B}(q, r_1)$, then $g_j(p^*) = g_j(q)$ for some $j = 1, \ldots, l$.

P2 The total number of blocks pointed to by q and containing only points from P' is less than cl.

Assume that $\mathcal{H}$ is a (r_1, r_2, p_1, p_2)-sensitive family; define $\rho = \frac{\ln 1/p_1}{\ln 1/p_2}$. The correctness of the LSH algorithm follows from the following theorem.

Theorem 1 *Setting $k = \log_{1/p_2}(n/B)$ and $l = \left(\frac{n}{B}\right)^{\rho}$ guarantees that properties* **P1** *and* **P2** *hold with probability at least $\frac{1}{2} - \frac{1}{e} \geq 0.132$.*

Remark 1 *Note that by repeating the LSH algorithm $O(1/\delta)$ times, we can amplify the probability of success in at least one trial to $1-\delta$, for any $\delta > 0$.*

Proof: Let property **P1** hold with probability P_1, and property **P2** hold with probability P_2. We will show that both P_1 and P_2 are large. Assume that there exists a point p^* within distance r_1 of q; the proof is quite similar otherwise. Set $k = \log_{1/p_2}(n/B)$. The probability that $g(p') = g(q)$ for $p \in P - \mathcal{B}(q, r_2)$ is at most $p_2^k = \frac{B}{n}$. Denote the set of all points $p' \notin \mathcal{B}(q, r_2)$ by P'. The expected number of blocks allocated for g_j which contain *exclusively* points from P' does not exceed 2. The expected number of such blocks allocated for all g_j is at most $2l$. Thus, by the Markov inequality [35], the probability that this number exceeds $4l$ is less than $1/2$. If we choose $c = 4$, the probability that the property P2 holds is $P_2 > 1/2$.

Consider now the probability of $g_j(p^*) = g_j(q)$. Clearly, it is bounded from below by

$$p_1^k = p_1^{\log_{1/p_2} n/B} = (n/B)^{-\frac{\log 1/p_1}{\log 1/p_2}} = (n/B)^{-\rho}.$$

By setting $l = \left(\frac{n}{B}\right)^{\rho}$, we bound from above the probability that $g_j(p^*) \neq g_j(q)$ for all $j = 1, \ldots, l$ by $1/e$. Thus the probability that one such g_j exists is at least $P_1 \geq 1 - 1/e$.

Therefore, the probability that both properties P1 and P2 hold is at least $1 - [(1-P_1) + (1-P_2)] = P_1 + P_2 - 1 \geq \frac{1}{2} - \frac{1}{e}$. The theorem follows. □

In the following we consider the LSH family for the Hamming metric of dimension d' as specified in Fact 2. For this case, we show that $\rho \leq \frac{1}{1+\epsilon}$ assuming that $r < \frac{d'}{\ln n}$; the latter assumption can be easily satisfied by increasing the dimensionality by padding a sufficiently long string of 0s at the end of each point's representation.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.