Journal of Pattern Recognition Research 1 (2014) 111-122

Received June 11, 2014. Revised October 15, 2014. Accepted October 28, 2014.

# Randomized Algorithm for Approximate Nearest Neighbor Search in High Dimensions

**Ruben Buaba**[1] *rbuaba@ncat.edu*

**Abdollah Homaifar**[1] *homaifar@ncat.edu*

**William Hendrix**[2] *whendrix@eecs.northwestern.edu*

**Seung Woo Son**[2] *sson@eecs.northwestern.edu*

**Wei-keng Liao**[2] *wkliao@eecs.northwestern.edu*

**Alok Choudhary**[2] *choudhar@eecs.northwestern.edu*

[1] *Electrical and Computer Engineering Department, North Carolina Agricultural and Technical State University, Greensboro, NC, USA*

[2] *Electrical Engineering and Computer Science Department, Northwestern University, Evanston, IL, USA*

## Abstract

A randomized algorithm employs a degree of randomness as part of its logic with uniform random bits as an auxiliary input to guide its behavior, in the hope of achieving good average runtime performance over all possible choices of the random bits. In this paper, we formulate a randomized algorithm capable of finding approximate nearest neighbors, specifically in high dimensional datasets. The random bits of this algorithm are sets of kernels chosen from the Gaussian normal distribution, which are used to create a data structure that guarantees sublinear runtime complexity and retrieval accuracy. The algorithm focuses on selecting the optimal cardinalities of the kernels and their members. These cardinalities influence computational, memory and runtime complexities of the algorithm. We demonstrate that using the cardinality of the kernels to determine the kernel size guarantees the highest provable lower bound of the probability of collision of related items; thus accurate retrieval of nearest neighbors. When implemented on Cray XE6 machine using 76 processes, our algorithm achieves speedup of 69 and approximately 48x performance gain in average query runtime with the retrieval accuracy of 90.5%.

*Keywords:* Approximate Nearest Neighbor, Failure Probability, Kernels, Linear Search, Locality Sensitive Hashing

## 1. Introduction

A nearest neighbor (NN) search can be formulated as follows: given a set $S$ of $n$ data points in a metric space $\mathcal{P}$, the task is to preprocess these points so that, given any query point $p \in \mathcal{P}$, the data point closest to $q$ can be retrieved quickly. Even though the linear search (LS) algorithm guarantees the retrieval of the nearest data point to a given query point correctly, it becomes computationally expensive and the runtime complexity can be extremely high when the dataset contains several data points with high dimensions (*as is the case in real life*). The reason being, LS literally iterates through the entire dataset in order to find the data point closest to a given query. A number of algorithms have been proposed which provide relatively modest constant factor improvements. Several authors suggested that projecting data points onto a single line results in faster query response time [1–3]. It is proposed that a prior partial distance computation could help [4]. Several articles on nearest neighbor search in datasets exist [5–9]. For uniformly distributed data points, expected query runtime is achievable by algorithms that decompose the search space into regular grids [10, 11]. Friedman, et al. generalized these results and reported that $\mathcal{O}(n)$ space and $\mathcal{O}(\log n)$ query time are possible using kd-trees [12]. However, even these methods suffer as dimensionality increases because the constant factors hidden in the asymptotic query runtime grow rapidly.

In reality, the dataset for most applications is dynamic - the dataset grows as new data is collected. Consequently, similarity search algorithms should scale to produce output within a reasonable timespan irrespective of the growth of the dataset. Thus, scalability with sublinear runtime complexity is paramount. Similarity search is a key component for many application areas, including but not limited to image and video database retrieval, data compression, information retrieval, pattern recognition, statistics and data analysis, and knowledge discovery and data mining [13, 14].

Over the years, intensive research has been conducted either to use trees, K-means clustering/classification or hashes to develop a space-partitioned data structure that would have a sublinear runtime [15]. Unfortunately, it is shown both theoretically and empirically that these solutions provide little or no improvement over the LS algorithm for large high dimensional dataset [16, 17]. Consequently, several researchers have become proponents of the use of approximate nearest neighbor (ANN) search algorithms [16–22]. The fundamental idea being, in practice, an ANN is almost as good as its exact nearest neighbor counterpart in most cases. Since a distance measure is what is often used for estimating how close two data points are, a small deviation in the distance should not adversely influence the similarity estimation unless the nearest neighbor problem itself is unstable [23, 24]. This notion of approximation forms the basis of a novel similarity search algorithm known as the Locality Sensitive Hashing (LSH), which was first introduced in [20]. LSH falls under randomized algorithms that use random input to reduce the expected runtime and memory usage, but always terminate with a correct result in a bounded amount of time. This technique drastically reduces the runtime complexity at the expense of a small probability of failing to find the absolute closest match.

Other algorithms that utilize randomness include hyper-encryption, Bayesian networks, random neural networks and probabilistic cellular automata. Quick sort is a familiar, commonly used algorithm in which randomness can be useful. Any deterministic version of this algorithm requires $\mathcal{O}(n^2)$ time to sort $n$ numbers for some well-defined class of degenerate inputs [25]. However, if the algorithm selects pivot elements uniformly at random, it has a provably high probability of sorting the $n$ numbers in $\mathcal{O}(n \log n)$ time regardless of the characteristics of the input.

Another example can be found in computational geometry, where a standard technique to build a structure like a convex hull or Delaunay triangulation using a technique known as randomized incremental construction is to randomly permute the input points and then insert them one by one into the existing structure [25]. The randomization ensures that the expected number of changes to the structure caused by an insertion is small. Thus, the expected running time of the algorithm can be bounded. This paper provides optimal parameterization of the LSH that guarantees the lowest provable runtime complexity and memory usage.

### 1.1 Paper Organization

The remainder of this paper is organized as follows: next sub-section formulates the problem; section 2 offers the background of LSH in general; section 3 explains theorems and

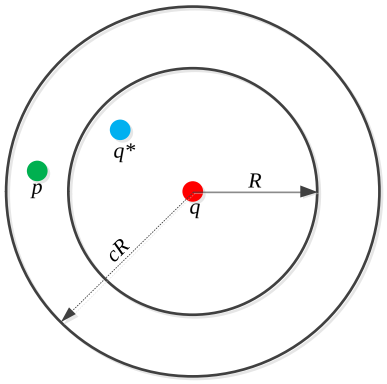

**Fig. 1:** R-c NN.

parametric constraints of the proposed technique; section 4 discusses the complexity of LSH; section 5 offers experimental evaluation of LSH on real data; and section 6 draws the conclusion.

### 1.2 Notations and Problem Definition

Unless otherwise stated, the following parameter notations are used throughout this paper. $\mathcal{P}$ is a set of $n$ data points in $d$-dimensional space $\Re^d$. A query data point and any other point are denoted by $q$ and $p$ respectively such that $p \neq q \in \mathcal{P}$. A sphere of radius $R$ centered at $q$ is denoted by $\beta(q, R)$. For any $\varepsilon > 0$, $c = 1 + \varepsilon$ is the approximation factor such that $cR > R$. The $l_2$ *norm* of $p$ is denoted by $\|p\|_2$. The expected number of data points per bucket (i.e. load factor) is denoted by $\alpha$. The goal is to select the cardinality of random Gaussian kernels (denoted by $L$) and the size of vector fields (denoted by $k$) in each kernel that would guarantee optimal runtime and memory space of LSH algorithm to find $p$ from $\mathcal{P}$ given $q$.

In other words, this algorithm is to solve the $(R, c)$-nearest neighbor problem in the $l_2$ *norm* space defined as follows (see Fig. 1): if $\exists\ q^* : \|q - q^*\|_2 \leq R$ then in sublinear runtime, report any point $p : \|q - p\|_2 \leq cR$, if it exists.

## 2. LSH Background

Locality Sensitive Hashing (LSH) is an index-based data structure that allows spatial item retrieval over a large dataset. The basic idea underlying the operation and the effectiveness of the LSH is that the algorithm is able to organize the data points in a given dataset into small clusters (buckets) by using random distance-preserving projections. In other words, distance between two data points $p, q \in \Re^d$ (i.e. $d \in Z^+$) in high-dimensional space is preserved in lower dimension with some small failure probability, $\delta$, to do so [26].

Suppose the real number line is chopped into segments numbered $0, 1, \ldots, m-1$ to form a table with $m$ slots. If integer values are assigned to the data points based on which segment they project to, then intuitively, making several projections certainly increases the probability, $P_1$ of nearby (local) points projecting to the same slot and decreases the probability, $P_2$ of remote points from projecting to the same slot. To further guarantee this, several sets of random vectors (hereafter referred to as hash functions) are used to perform the scalar projections. Whenever two or more items hash into the same bucket on any table, collisions occur, and these can be resolved using double hashing, a linked list, or chaining. A well-developed hash function tries to amplify the gap between the two probabilities, $P_1$ and $P_2$. To achieve this goal, the hash functions must be locality sensitive and universal [27].

**Definition 1.** *A family of hash functions, $\mathcal{H} = \{h_{ij} \in h_j : \mathcal{P} \to U\}$ each $h_{ij}$ is a vector field from kernel $h_j$ mapping one point from domain $\mathcal{P}$ to domain $U$, is said to be $P_1 > P_2$, $cR > R$ locality-sensitive if for any $p, q \in \mathcal{P}$:*

- *if $p \in \beta(q, R)$, $\Pr[h_{ij}(p) = h_{ij}(q)] \geq P_1$('high')*
- *if $p \notin \beta(q, cR)$, $\Pr[h_{ij}(p) = h_{ij}(q)] \leq P_2$('low')*

On the other hand, suppose $\mathcal{P}$ is a universe of keys, and $\mathcal{H}$ is a family of a finite collection of hash functions, each mapping $U$ to $0, 1, \ldots, m-1$, $\mathcal{H}$ is said to be universal if $\forall p, q \in \mathcal{P}$ and $p \neq q$, $\Pr[h_{ij} \in \mathcal{H} : h_{ij}(p) = h_{ij}(q)] = \mathcal{H}/m$.

Creation of hash tables is normally fast and simple if the objects in the dataset are binary strings (i.e. $\{0, 1\}^d$) [16, 20] - the biggest drawback of LSH. In spite of this limitation, LSH has been used in a number of applications involving non-binary datasets [28–35]. To handle non-binary datasets, the algorithm has to be extended to the $l_2$ *norm* by embedding $l_2$ space into $l_1$ space, and then $l_1$ space into the binary Hamming space. For a dynamic dataset, the major advantage of LSH over tree-based data structures is its ability to support deletion and insertion operations with $\mathcal{O}(1)$ time complexity [15].

Once the hash table is created and stored, for any query point $q \in \mathcal{H}$, the $(R, c)$-NNs can be found by hashing $q$ and retrieving data points stored in the buckets $h_1(p)$, $h_2(p)$,..., $h_L(p)$ into which the query point has hashed. Therefore, when the load factor varies, there is a trade-off between a larger table with a smaller final linear search or a more compact table with more data points to consider in the final search. The $(R, c)$-NN search is terminated after finding the first $2L$ data points (including duplicates) closet to $q$ [15]. The parameters $k$ and $L$ are chosen such that the following two conditions hold with constant probabilities:

**Lemma 1.** *If $q^* \in \beta(q, R)$, then $h_j(q^*) = h_j(q)$ for some $j \in \{1, 2, \ldots, L\}$.*

**Lemma 2.** *The expected number of collision of $q$ with any point $q \in \mathcal{P}$ such that $p \in \beta(q, cR)$ is less than $2L$.*

## 3. Theory and Constraints

The LSH algorithm works with some basic theorems and parametric constraints. These theorems and constraints are analyzed.

### 3.1 $s$-Stable Distribution

Stable distributions are defined as limits of normalized sums of independent identically distributed variables [36].

**Theorem 1.** *A distribution $D$ is said to be $s$-stable if there exists $s \geq 0$, such that for any $N$ real numbers $u_1, u_2, \ldots, u_N$ and independent identically distributed random variables, $X_1, X_2, \ldots, X_N$ with distribution $D$, the random variable, $\sum_{i=1}^{N} u_i X_i$ has the same distribution as the variable $\left(\sum_{i=1}^{N} |u_i|^s\right)^{1/s}$, where $X$ is a random variable with distribution $D$.*

For $s \in [0, 2]$, there exist stable distributions [36]. However, the focus is shifted to the case where $s = 2$ since the distance measure normally used is the $l_2$ *norm* (the Euclidean space).

Under this choice, the normal Gaussian distribution denoted by $\mathcal{N}(0,1)$ with a probability distribution function, $f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$ is 2-stable.

Suppose a random vector $h_{ij} \in \Re^d$ is chosen from the standard Gaussian distribution, $h_{ij} \sim \mathcal{N}(0,1)$ and $p$ is any vector such that $p \in \Re^d \rightarrow p = [p_1, p_2, \ldots, p_d]$. Then, the scalar dot product $h_{ij} \cdot p$ is a random variable which tends to be distributed as $\|p\|_2 h_{ij}$, where $h_{ij}$ is a random variable with 2-stable distribution and $\|p\|_2$ is the $l_2$ *norm* of vector $p$ given as $\|p\|_2 = \left(\sum_{z=1}^{d} p_z^2\right)^{1/2}$. Thus, the difference, $h_{ij} \cdot p - h_{ij} \cdot q$ tends to be distributed as $\|q-p\|_2 h_{ij}$. Small collections of such dot products corresponding to different $h_{ij} \in h_j$ can be used to estimate $\|q-p\|_2$, the $l_2$ *norm* between the two data points $p$ and $q$. Thus, the LSH scheme is said to be $l_2$-embedding and the $l_2$ *norm* forms the similarity metric for finding the nearest neighbors of a given query data point.

### 3.2 Evaluating Prior Probabilities $P_1$ and $P_2$

Suppose $c = \|q-p\|_2$ and $b_{ij}$ is an offset drawn uniformly from $[0, \alpha]$ at random. Also assume that $h_j$ is a family of $k$ hash functions drawn randomly and independently from $\mathcal{N}(0,1)$ such that $h_j = [h_{1j}, h_{2j}, \ldots, h_{kj}]$. The hash value of $p$ can be computed as

$$h_j(p) = \left[\left\lfloor\frac{h_{1j} \cdot p + b_{1j}}{\alpha}\right\rfloor, \left\lfloor\frac{h_{2j} \cdot p + b_{2j}}{\alpha}\right\rfloor, \ldots, \left\lfloor\frac{h_{kj} \cdot p + b_{kj}}{\alpha}\right\rfloor\right]$$

From the 2-stable distribution, the difference, $h_{ij} \cdot p - h_{ij} \cdot q$ for any $i^{\text{th}}$ hash function has the same distribution as $ch_{ij}$ . The probability, $\Pr(c)$ that $p$ collides with $q$ is given by equation (1). The $F(\cdot)$ in equation (1) represents the probability density function of the absolute value of the Gaussian distribution.

$$\Pr(c) = \Pr[h_{ij}(p) = h_{ij}(q)] = \int_0^{\alpha} \frac{1}{c} F\left(\frac{t}{c}\right)\left(1 - \frac{t}{\alpha}\right) dt \tag{1}$$

It should be noted that for a given $\alpha$, the probability of collision decreases monotonically with $c$. This means that the probability of collision is high if $\|p-q\|_2$ is small and low if $\|p-q\|_2$ is large respectively. Thus, as per the definition of locality-sensitivity, for this to be $P_1 > P_2$, $c$-sensitive, $P_1 = \Pr(c=1)$ and $P_2 = \Pr(c>1)$. Solving this yields equations (2) and (3) with $F_{\text{cdf}}(\cdot)$ being the cumulative distribution function of the Gaussian random variable.

$$P_1 = 1 - 2F_{\text{cdf}}(-\alpha) - \frac{2}{\sqrt{2\pi}\alpha}\left(1 - e^{-\alpha^2/2}\right) \tag{2}$$

$$P_2 = 1 - 2F_{\text{cdf}}(-\alpha/c) - \frac{2}{\sqrt{2\pi}\alpha/c}\left(1 - e^{-\alpha^2/2c^2}\right) \tag{3}$$

These prior probabilities influence the performance measure $\rho = \ln P_1 / \ln P_2$ (i.e. amplification gap), which is obtained by selecting $\alpha$ that minimizes this ratio at a specific $c$. An approximate solution is given as $\rho(c) \approx 1/c$ by [20] $\alpha = 4$. However, we show that $\alpha = 5$ gives a better optimal value for $\rho$ than that of $\alpha = 4$ [37].

### 3.3 Cardinalities of Kernels and Kernel Size

The number of kernels ($L$) and the kernel size ($k$) play key roles in the performance of the LSH algorithm. From equation (2), the probability of collision for a single scalar projection is $P_1$. Let $\delta$ be the probability of false negatives; a typical choice of $\delta$ is 0.1 [15]. Thus, the probability of no collision under all $k$ projections is given by $1 - (P_1)^k$. This means that the probability of no collision under all $L$ tables is $\left(1-(P_1)^k\right)^L$. Thus, $\left(1-(P_1)^k\right)^L \geq \delta$, which yields equation (4).

$$k \leq \ln(1-\delta^{1/L})/\ln P_1 < \log_{1/P_2} n \tag{4}$$

Now, the only remaining variable is $L$ and this is chosen as in equation (5).

$$L = n^{\rho} \tag{5}$$

For a fixed $i$ and $j$, the probability of collision under all $k$ projections is $\Pr[h_{ij}(q^*) = h_{ij}(q)] \geq (P_1)^k$. Thus, $(P_1)^k$ is expressed from equations (4) and (5) by equation (6):

$$(P_1)^k = (P_1)^{\ln(1-\delta^{1/n^{\rho}})/\ln P_1} = 1 - \delta^{1/n^{\rho}} \tag{6}$$

Now, the probability that $q^*$ collides with $q$ for some $i$ and $j$ under the $L$ tables is given by equation (7), which, when evaluated, results in equation (8).

$$\Pr[\exists\, i,j : h_{ij}(q^*) = h_{ij}(q)] \geq 1 - \left(1 - \left(1 - \delta^{1/n^{\rho}}\right)\right)^{n^{\rho}} \tag{7}$$

$$\Pr[\exists\, i,j : h_{ij}(q^*) = h_{ij}(q)] \geq 1 - \delta \tag{8}$$

To the best of our knowledge, this is the highest provable lower bound of the probability of collision. For the widely used $\delta = 0.1$, the probability of collision is at least 90%.

Suppose $q^* \in \mathcal{P}$ such that $\|q - q^*\|_2 > cR$. Then the probability of collision under all $k$ projections is given by equation (9). Substituting for $k$ from equation (4) yields equation (10).

$$\Pr[h_{ij}(q^*) = h_{ij}(q)] \leq (P_2)^k \tag{9}$$

$$(P_2)^k = (P_2)^{\log_{1/P_2} n} = 1/n \tag{10}$$

Consequently the expected number of collision is at most unity and the total expected number of collision on all $L$ tables is $L$. From the Markov's inequality theorem, the probability that $q^*$ collides with more than $2L$ data points is given as

$$\Pr\left[E\left(h_{ij}(q^*) = h_{ij}(q)\right) > 2L\right] < 1/2$$

Thus, these choices for $k$ and $L$ meet the stated conditions (see **Lemma 1** and **Lemma 2**). In [20], the authors outlined the traditional ways for choosing these critical parameters, $k$ and $L$. It is worth mentioning that our formulation cuts the $L$ by 1/2. Thus, causing approximately 50% reduction in computational and memory cost and improves the query runtime by a factor of 2.

## 4. Algorithm Complexity

The main goal of choosing optimal parameters for the LSH is to enable the fast computations of the hash values; and to use as little memory space as possible to store these hash values. These are necessary in order to guarantee fast query runtime. Building hash tables may take quite amount of time. As a result, they are created only once and stored along with their families of hash functions. The tables are then used to answer different query inputs in sub-linear runtime.

## 4.1 Hash Tables Building Steps

The steps for building the hash tables are listed below:

1. Generate $L$ families of hash functions, $h_1, h_2, \ldots, h_L$ randomly and independently from $\mathcal{N}(0,1)$ such that each $h_j = [h_{1j}, h_{2j}, \ldots, h_{kj}]$ and each hash function $h_{ij} \in \Re^d \ \forall\ j \in [1,2,\ldots,L]$ and $\forall\ i \in [1,2,\ldots,k]$.
2. Generate offset $b_{ij}$ randomly, independently and uniformly from $[0,\alpha]$ for each $i^{\text{th}}$ hash function and each $j^{\text{th}}$ family.
3. Generate $L$ set, $H_1, H_2, \ldots, H_L$ of random integers from the range $[1,m]$, independently such that each $H_j \in \Re^k$.
4. Index all the $n$ data points in $\mathcal{P}$ from 1 through $n$.
5. For each feature vector $p \in \mathcal{P}$, normalize $p$ as $\hat{p} = p/\|p\|_2$.
6. Compute the hash value for $\hat{p}$ for the $j^{\text{th}}$ family as
$$h_j(\hat{p}) = \left[\left\lfloor\frac{h_{1j}\cdot p + b_{1j}}{\alpha}\right\rfloor, \left\lfloor\frac{h_{2j}\cdot p + b_{2j}}{\alpha}\right\rfloor, \ldots, \left\lfloor\frac{h_{kj}\cdot p + b_{kj}}{\alpha}\right\rfloor\right]$$
7. Compute the second level hash value for $\hat{p}$ as $h_j^*(\hat{p}) = ((h_j(\hat{p})\cdot H_j) \mod M) \mod m$.
8. Store the index of $p$ in the bucket $h_j^*(\hat{p})$ on the $j^{\text{th}}$ table.

The $M$ is a large prime integer close to $2^W$, where $W$ is the word width of the microprocessor being used. For a 64-bit computer, $M = 2^{64} - 5$. In [38, 39], we offer more details regarding the formulation of the equation for computing the hash values.

## 4.2 Table Lookup (Querying) Steps

The steps for table lookup are listed below:

1. For a given query feature vector $q \in \mathcal{P}$ normalize $q$ (i.e. $\hat{q} = q/\|q\|_2$).
2. Compute the hash values for $q$ as $h_1(\hat{q}), h_2(\hat{q}), \ldots, h_L(\hat{q})$.
3. Compute the second level hash values as $h_1^*(\hat{q}), h_2^*(\hat{q}), \ldots, h_L^*(\hat{q})$.
4. Use the indices in these buckets to collect their corresponding feature vectors (let us call this the Candidate set).
5. Compute the $l_2$ *norm* between $q$ and the entries in the Candidate set.
6. Retrieve the top $K$, $(R,c)$-NNs to $q$ or terminate the search once $2L$ (including duplicates) items are retrieved, whichever comes first.

## 4.3 Computational Complexity

Suppose $h_{ij}$ is the $i^{\text{th}}$ hash function for the $j^{\text{th}}$ table. Assuming that it takes one computational operation to project a data point, $p$, in the direction of $h_{ij}$ (i.e. $h_{ij}\cdot p$). Making $k$ projections requires $k$ operations. To project $p$ onto all the $L$ tables requires $kL$ operations. Thus, to project all $n$ data points requires $nkL$ operations. As a result, the computational cost is $\mathcal{O}(nkL)$. The hidden constant in $\mathcal{O}(nkL)$ depends on the dimensionality of $p$. For a fixed $n$, the computational cost grows as a function of $k$ and $L$. This implies that both $k$ and $L$ must be at their minimum best to ensure low computational cost. The computational cost becomes $\mathcal{O}(n^{1+\rho}\ln(1-\delta^{1/n^{\rho}}))$.

### 4.4 Memory Space

To store each data point (i.e. $p \in \Re^d$) along with its $k$ concatenated hash values on all $L$ tables requires $L(d+k)$ memory. If the $k$-concatenated values are hashed to produce a single integer, then the memory requirement reduces to $L(d+1)$ per data point. For all $n$ data points, the memory requirement becomes $\mathcal{O}(dnL + nL)$. For a large dataset this can be huge.

In our implementation however, two optimization techniques are used to reduce the memory requirement further. First, all the $n$ data points in $\mathcal{P}$ are indexed from 1 through $n$. The indexing is the same across all the $L$ hash tables. This means that each data point is stored once instead of $L$ times. This reduces the memory to $\mathcal{O}(dn+nL)$. Second, the single integer produced from hashing the $k$-concatenated values is not stored but rather used to point to a bucket. The index of the data point is then stored in that bucket. Thus, each data point can be referred to by an index. The hidden constant in $\mathcal{O}(dn)$ depends on the data type of the data points in the dataset.

Henceforth, the memory analysis concentrates on the hash tables of the data structure. Therefore, the memory requirement to store the hash tables for all the $n$ data point is $\mathcal{O}(nL)$. Once again, $L$ has to be kept at its minimum best to achieve the best memory. The hidden constant in $\mathcal{O}(nL)$ is the number of bytes required to point to each bucket and $0.125\lceil\log_2 n\rceil$ bytes to store each index of the data point on each table. Thus, the memory requirement becomes $\mathcal{O}(n^{1+\rho})$

### 4.5 Query Runtime

When running a query $q$, two time complexities are involved. First, the time required to hash the query to a bucket on each of the $L$ tables to retrieve the candidate set. Second, the time required to compute the distance between the query and the entries in the candidate set. Let $\tau_h$ and $\tau_c$ denote these respectively. Computing the hash value $h_j(\hat{p})$ for all the tables is $\mathcal{O}(kL)$. The second level hash value computation is $\mathcal{O}(L)$ and it is relatively insignificant compared to that of the first. Thus, $\tau_h$ is $\mathcal{O}(kL)$. Suppose it takes one computational operation to compute the $l_2$ *norm* between $q$ and an entry in the candidate set. The expected number of entries in the candidate set is $\alpha L$. Thus, $\tau_c$ becomes $\mathcal{O}(\alpha L)$. As a result, the total query runtime is $\mathcal{O}(kL + \alpha L)$.

If the $(R, c)$-NNs are to be sorted then the computed $l_2$ *norms* have to be sorted using a sorting algorithm such as a quicksort. Let this be denoted as $\tau_s$. The quicksort average computational cost is $\mathcal{O}(\alpha L \log_2 \alpha L)$. The total query runtime becomes $\tau_h + \tau_c + \tau_s$. The hidden constants in these analyses depend on the dimensionality and the complexity of the data points.

## 5. Experimental Evaluation

The algorithm discussed in this paper is data-independent. In other words, the average performance of the algorithm remains fairly constant regardless of the dataset being used. We evaluated our proposed scheme on NERSCs Hopper. Hopper is a Cray XE6 machine, with a peak performance of 1.28 Petaflops/sec. Each compute node in Hopper has 2 twelve-core AMD processors and 32 GB memory. The Hopper system has a locally attached high-performance disk spaces, each of 1 PB capacity. Our input and output files are stored on this scratch disk spaces. We implemented our proposed approach using the parallel netCDF 1.4.0 and Cray MPICH 6.0.1. Both input and output file is read and written in the netCDF format. We compile both LSH build and query programs using PGI compiler version 13.6.0 with the -fast compilation flag.

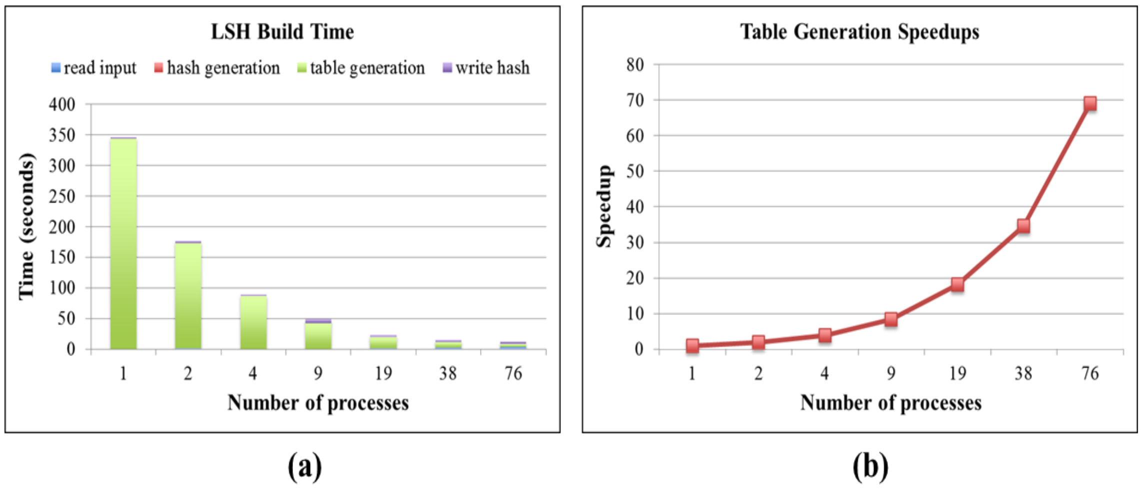

**Fig. 2:** (a) Breakdown of hash table building time. (b) Speedup for hash table generation.

## 5.1 Hash Table Creation

We first evaluated the performance of our hash building scheme for the texture feature vectors of the images. The following are the parameter values used for generating hash tables: $c = 3.3$, $\delta = 0.1$, $P_1 = 0.8404$, $P_2 = 0.5108$, $\rho = 0.2588$, $n = 1604950$, $\alpha = 5$, $L = 76$, and $k = 22$. It should be noted that the number of buckets (i.e. $m$) on each table is computed as a prime approximation of $\lceil n/\alpha \rceil$, where $n$ is the number of items in the dataset. Also, the ceiling operator $\lceil \cdot \rceil$ must be applied to the computation of both $k$ and $L$ since both have to be positive integers ($k, L \in Z^+$). In addition, we used a random number to seed the random number generator. Fig. 2 shows the breakdown of the hash build time while varying the number of processors up to 76. The input file contains about 1.6 million texture feature vectors of images extracted from Defense Meteorological Satellite Program satellite imagery archives (about 123MB). A detailed discussion of the texture feature extraction approach could be found in [40].

The generated hash table is about 557MB, about 4.5 times larger than the input file. Several observations can be made from the graphs. Since the input and output file sizes are small, the time spent on file I/O remained relatively constant. Also, the time spent on hash generation is almost negligible. The most important observation from these results is that the table generation does scale as we increased the number of processors, for a speedup of approximately 69 on 76 processes.

## 5.2 Querying

We next performed query operations on the hash tables built from our previous experiments. The hash tables built from the previous experiments are to be used to find the nearest neighbors to any given query chosen from the dataset by searching a fraction of the entire database. We randomly selected 10 query vectors from the entire dataset (DS). Table 1 shows the comparison between LSH and linear search (LS) schemes in terms of the average query runtime and the percentage of dataset scanned for the 10 queries to report the nearest neighbors.

Both LSH and LS report the top 100 nearest neighbors. We observed that LSH achieved almost $48\times$ performance gain in average query runtime with the accuracy of 90.5%.

**Table 1:** Comparison between LSH and LS (Yardstick)

| Scheme | Average | Runtime | Gain | Accuracy |
|---|---|---|---|---|
| LSH | 0.02 s | 47.60 | 90.50% | 1.31% |
| LS | 0.96 s | NA | NA | 100% |

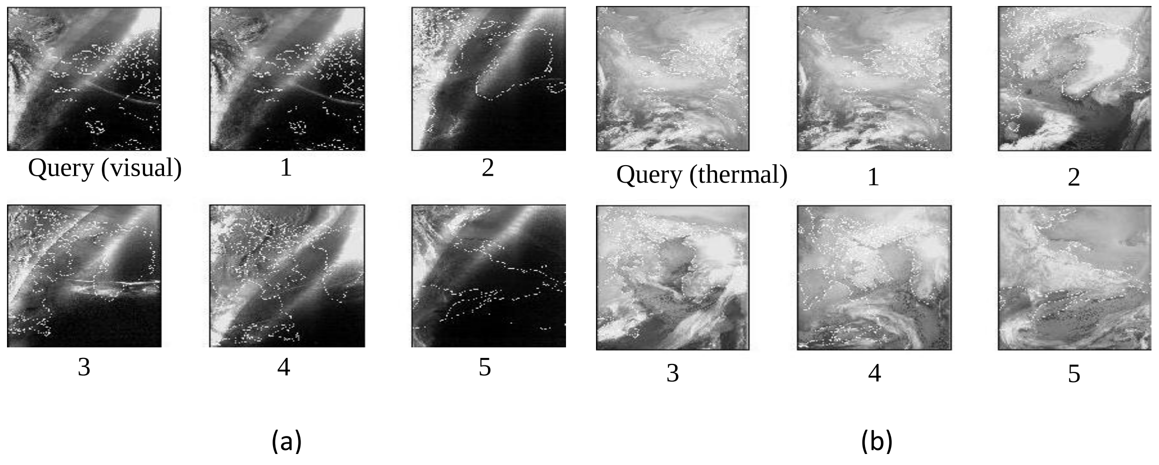

**Fig. 3:** (a) Visual query image and its top five similar matches. (b) Thermal query image and its top five similar.

For visual purposes, Fig. 3 shows one sample query image and its top five similar visual and thermal images respectively. The images are ranked: ‘1’ being the best similar image and ‘5’ being the worst similar image. To understand the gain by LSH in detail, we next measured the number of images compared in LSH. Note that, the LS scheme needs to search all the dataset, or about 1.6 million items. LSH, on the other hand, compared the query with only 1.31% of the entire dataset.

## 6. Conclusion

In a large dataset retrieval applications in which approximate matches are acceptable, LSH is very effective. Unlike linear search (LS), hash tables need to be built when using LSH and this takes time, but once this is done, LSH far surpasses LS in terms of the query runtime complexity. For implementation purposes, each hash table is only flushed to a disk after computing its hash values. LSH aims to search a fraction of the dataset to find nearest neighbors. Thus, LSH supports sublinear runtime and scalability to larger datasets. The number of tables created and the number of projections used have significant effect on the performance of the LSH. These variables are related and are dependent on size of the dataset.As has been shown in this paper, using the number of kernels to compute the size of vector fields in each kernel provides the highest provable probability of collision of related items under the LSH algorithm. Further analyses also show that, this approach achieves lower computational, lower memory and faster query runtime complexities than the traditional method, where the number of kernels is rather determined by the size of the kernel to use. The ability to create the hash tables independently enables LSH to use parallel computing resources for faster table creation as we have shown. This algorithm is extendable to all other fractional norms aside the Euclidean norm.

## Acknowledgments

The authors are grateful to National Science Foundation (NSF); and National Oceanic and Atmospheric Administration/National Geophysical Data Center (NGDC) in Boulder, Colorado, for providing us with the DMSP satellite imagery dataset; and to the Expeditions in Computing by the National Science Foundation (NSF) for their support.

## References

[1] J. H. Friedman, F. Baskett, and L. J. Shustek, *An algorithm for finding nearest neighbors*, Computers, IEEE Transactions on, vol. C-24, pp. 1000-1006, 1975.

[2] L. Guan and M. Kamel, *Equal-average hyperplane partitioning method for vector quantization of image data*, Pattern Recognition Letters, vol. 13, pp. 693-699, 1992.

[3] C. H. Lee and L. H. Chen, *Fast closest codeword search algorithms for vector quantization*, Vision, Image and Signal Processing, IEE Proceedings, vol. 141, pp. 143-148, 1994.

[4] B. Chang-Da and R. Gray, *An improvement of the minimum distortion encoding algorithm for vector quantization*, Communications, IEEE Transactions on, vol. 33, pp. 1132-1133, 1985.

[5] K. I. Lin, H. V. Jagadish, and C. Faloutsos, *The TV-tree: an index structure for high-dimensional data*, The VLDB Journal, vol. 3, pp. 517-542, 1994.

[6] N. Roussopoulos, S. Kelley, and F. Vincent, *Nearest neighbor queries*, in Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, United States, 1995.

[7] D. A. White and R. Jain, *Similarity indexing with the SS-tree*, in Proceedings of the Twelfth International Conference on Data Engineering, pp. 516-523, 1996.

[8] S. Berchtold, D. A. Keim, and H.-P. Kriegel, *The X-tree: an index structure for high-dimensional data*, in Proceedings of the 22th International Conference on Very Large Data Bases, 1996.

[9] S. Berchtold, C. Bhm, D. A. Keim, and H.-P. Kriegel, *A cost model for nearest neighbor search in high-dimensional data space*, in Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona, United States, 1997.

[10] J. G. Cleary, *Analysis of an algorithm for finding nearest neighbors in Euclidean space*, ACM Trans. Math. Softw., vol. 5, pp. 183-192, 1979.

[11] J. L. Bentley, B. W. Weide, and A. C. Yao, *Optimal expected-time algorithms for closest point problems*, ACM Trans. Math. Softw., vol. 6, pp. 563-580, 1980.

[12] J. H. Friedman, J. L. Bentley, and R. A. Finkel, *An algorithm for finding best matches in logarithmic expected time*, ACM Trans. Math. Softw., vol. 3, pp. 209-226, 1977.

[13] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, H. Qian, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker, *Query by image and video content: the QBIC system*, Computer, vol. 28, pp. 23-32, 1995.

[14] U. M. Fayyad, *Advances in knowledge discovery and data mining*, AAAI Press, 1996.

[15] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, *Locality-sensitive hashing scheme based on p-stable distributions*, in Proceedings of the Twentieth Annual Symposium on Computational Geometry, Brooklyn, New York, USA, 2004.

[16] 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, 1999.

[17] 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 the 24-th International Conference on Very Large Data Bases, 1998.

[18] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu, *An optimal algorithm for approximate nearest neighbor searching*, in Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, Arlington, Virginia, United States, 1994.

[19] S. Har-Peled, *A replacement for Voronoi diagrams of near linear size*, in 42nd IEEE Symposium on Foundations of Computer Science, pp. 94-94, 2001.

[20] P. Indyk and R. Motwani, *Approximate nearest neighbors: towards removing the curse of dimensionality*, in Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, Dallas, Texas, United States, 1998.

[21] J. M. Kleinberg, *Two algorithms for nearest-neighbor search in high dimensions*, in Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of computing, El Paso, Texas, United States, 1997.

[22] E. Kushilevitz, R. Ostrovsky, and Y. Rabani, *Efficient search for approximate nearest neighbor in high dimensional spaces*, in Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, Dallas, Texas, United States, 1998.

[23] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, *When is "Nearest Neighbor" meaningful?*, in Proceedings of the 7th International Conference on Database Theory, 1999.

[24] A. Hinneburg, C. C. Aggarwal, and D. A. Keim, *What Is the Nearest Neighbor in high dimensional spaces?*, in Proceedings of the 26th International Conference on Very Large Data Bases, 2000.

[25] R. Seidel, *Backwards analysis of randomized geometric algorithms*, in New Trends in Discrete and Computational Geometry, vol. 10, J. Pach, Ed., ed: Springer Berlin Heidelberg, pp. 37-67, 1993.

[26] M. Slaney and M. Casey, *Locality-sensitive hashing for finding nearest neighbors*, Lecture Notes, Signal Processing Magazine, IEEE, vol. 25, pp. 128-131, 2008.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Second Edition, MIT Press, 2001.

[28] J. Buhler, *Efficient large-scale sequence comparison by locality-sensitive hashing*, Bioinformatics, vol. 17, pp. 419-28, 2001.

[29] J. Buhler, *Provably sensitive indexing strategies for biosequence similarity search*, in Proceedings of the Sixth Annual International Conference on Computational Biology, Washington, DC, USA, 2002.

[30] J. Buhler and M. Tompa, *Finding motifs using random projections*, J Comput. Biol., vol. 9, pp. 225-42, 2002.

[31] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang, *Finding interesting associations without support pruning*, in IEEE Transactions on Knowledge and Data Engineering, vol. 13, pp. 64-78, 2001.

[32] B. Georgescu, I. Shimshoni, and P. Meer, *Mean shift based clustering in high dimensions: a texture classification example*, in Proceedings of the Ninth IEEE International Conference on Computer Vision, vol. 1, pp. 456-463, 2003.

[33] Z. Ouyang, N. D. Memon, T. Suel, and D. Trendafilov, *Cluster-based delta compression of a collection of files*, in Proceedings of the 3rd International Conference on Web Information Systems Engineering, 2002.

[34] Y. Cheng, *MACS: music audio characteristic sequence indexing for similarity retrieval*, in Proceedings of the 2001 IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics, pp. 123-126, 2001.

[35] J. Nolan, *Stable distributions: models for heavy-tailed data*, Springer Verlag, 2007.

[36] V. M. Zolotarev, *One-dimensional stable distributions*, American Mathematical Society, 1986.

[37] R. Buaba, A. Homaifar, and E. Kihn, *Optimal load factor for approximate nearest neighbor search under exact Euclidean locality sensitive hashing*, International Journal of Computer Applications, vol. 69, pp. 22-31, May, 2013.

[38] R. Buaba, A. Homaifar, M. Gebril, and E. Kihn, *Satellite image retrieval application using Locality Sensitive Hashing in L2-space*, in Proceedings of the 2011 IEEE Aerospace Conference, pp. 1-7, 2011.

[39] R. Buaba, A. Homaifar, M. Gebril, E. Kihn, and M. Zhizhin, *Satellite image retrieval using low memory locality sensitive hashing in Euclidean space*, Earth Science Informatics, vol. 4, pp. 17-28, 2011.

[40] M. Gebril, R. Buaba, A. Homaifar, and E. Kihn, *Structural indexing of satellite images using automatic classification*, in Proceedings of the 2011 IEEE Aerospace Conference, pp. 1-7, 2011.