PAPER

Approximate Nearest Neighbor Search for a Dataset of Normalized Vectors

Kengo TERASAWA[†a)], ***Member and*** **Yuzuru TANAKA**[††], ***Nonmember***

SUMMARY This paper describes a novel algorithm for approximate nearest neighbor searching. For solving this problem especially in high dimensional spaces, one of the best-known algorithm is Locality-Sensitive Hashing (LSH). This paper presents a variant of the LSH algorithm that outperforms previously proposed methods when the dataset consists of vectors normalized to unit length, which is often the case in pattern recognition. The LSH scheme is based on a family of hash functions that preserves the locality of points. This paper points out that for our special case problem we can design efficient hash functions that map a point on the hypersphere into the closest vertex of the randomly rotated regular polytope. The computational analysis confirmed that the proposed method could improve the exponent ρ, the main indicator of the performance of the LSH algorithm. The practical experiments also supported the efficiency of our algorithm both in time and in space.

key words: *nearest neighbor, randomized algorithm, locality-sensitive hashing (LSH)*

1. Introduction

This paper describes a novel algorithm for approximate nearest neighbor searching. The proposed algorithm is a variant of the LSH (Locality-Sensitive Hashing) algorithm and it works more efficiently than other known algorithms for the case when the dataset consists of vectors normalized to unit length.

Given a set of points in a metric space and a query point, the nearest neighbor search is the problem of finding a point in the dataset that is closest to the query point. It is one of the most fundamental problems in computer science, with many applications in information retrieval, pattern recognition, clustering, machine learning, data mining, and so forth. In pattern recognition, for example, the most common way of similarity search is to represent the feature of the objects as the multi-dimensional vector and find the nearest neighbor in the vector space. Such applications include content-based image retrieval, document retrieval, face recognition, fingerprint recognition, etc.

An infallible way to find the nearest neighbor is to compute the distance from the query point to every other point in the dataset. This exhaustive search, also called brute-force search, can solve the problem in $O(dn)$ time, where n is the population of the given dataset and d is the dimensionality of the data. Nowadays, since the size of the data we should treat tends to become larger and larger, the linearity of the complexity to the size has posed a problem. Therefore, it is important to devise an algorithm that solves the nearest neighbor problem faster than the brute-force method. As a result of intensive research efforts, some data structures such as kd-tree [1] gave us a good solution to this problem, particularly for low dimensional spaces.

However, no algorithm has provided a good solution to this problem yet when the dimensionality goes higher. For all known algorithm, it happens that either the time complexity asymptotically tends to be $O(dn)$, which means no improvement over the brute-force method, or the required memory space is exponential to d, which is of course infeasible when d is large. Such a difficulty, known as "the curse of dimensionality," poses a problem especially in pattern recognition applications where the dimensionality tends to be tens to hundreds or more.

To overcome the curse of dimensionality, the approximated methods have gotten a lot of attention recently. The c-approximate nearest neighbor problem is the relaxed problem that allows an output point to be at most c times distant than the exact nearest neighbor is. This $c > 1$ is called the approximation factor. A randomized (or probabilistic) algorithm is also often employed to overcome the curse of dimensionality. With a fixed parameter δ, the randomized algorithm should return requested point with a probability of no less than $1 - \delta$. Using these approximations, many algorithms have been proposed. Among them, one of the best-known algorithms is Locality-Sensitive Hashing.

Locality-Sensitive Hashing (LSH) [2]–[5] is a randomized algorithm for the approximate nearest neighbor problem. It is proved that LSH can solve the approximate nearest neighbor problem in less than $O(dn)$ time, i.e., sublinear time. The key idea of LSH is to map each point in the dataset into a certain bucket using a hash function randomly chosen from the function family called LSH family. In finding the nearest neighbor, LSH examines only the points that are mapped to the same bucket as the query point is mapped to. As a result, the query time of LSH is dominated by $O(n^\rho)$ distance computations and $O(n^\rho \log_{1/p_2} n)$ evaluations of hash functions (the detailed meaning of p_2 and ρ will be described later).

The performance of LSH has been improved several times during this decade. One such improvement was the expansion of the applicability to various metric spaces.

Manuscript received October 31, 2008.
Manuscript revised April 22, 2009.

[†]The author is with Future University-Hakodate, Hakodate-shi, 041–8655 Japan, and PRESTO, Japan Science and Technology Agency, Kawaguchi-shi, 332–0012 Japan.

[††]The author is with Meme Media Laboratory, Hokkaido University, Sapporo-shi, 060–8628 Japan.

a) E-mail: kterasaw@fun.ac.jp

DOI: 10.1587/transinf.E92.D.1609

While the originally proposed LSH [2] was basically applicable only to metric space under l_1 norm, the later version [3] was applicable to metric space under l_s norm for any $s \in (0, 2]$. Another important improvement was the reduction of the exponent ρ. While the value of ρ was just $1/c$ in the original method, the Leech Lattice-based method proposed in [4] has reduced the value of ρ to 0.5563 for $c = 1.5$, and to 0.3641 for $c = 2.0$.

The aforementioned efforts to improve LSH had been devoted to rather general situation. In many cases of a pattern recognition applications, however, the nearest neighbor problem we face is sometimes more specific. The most commonly used metric is l_2 norm, i.e., Euclidean distance. Besides, the points in the dataset are often limited to lie on the surface of the unit hypersphere rather than distributed in the entire $\mathbb{R}^d$ space. This is because the direction of the feature vector is more important to discriminate the objects than the magnitudes of the vector. For example, the SIFT (Scale Invariant Feature Transform) descriptor [6], [7], which is one of the most famous descriptors in computer vision, uses a 128-dimensional feature vector with its length normalized to unity: it means that SIFT descriptors are distributed over a unit hypersphere embedded in $\mathbb{R}^{128}$ space. For another example, regarding the term frequency used in text processing, where the dimensionality of the feature tends to be hundreds to thousands or more, the most widely used distance metric is the cosine similarity [8]. Note that the nearest neighbor search with a cosine similarity measure is equivalent to the nearest neighbor search with a Euclidean distance measure after normalizing all the vectors to unit length. In addition to those mentioned here, we have many examples of pattern recognition applications that use vectors normalized to unit length.

Focusing on this fact, this paper proposes a novel variant of the LSH specifically designed for the case when all points in the dataset are limited to lie on the surface of the unit hypersphere. The proposed algorithm, named the Spherical LSH (SLSH), uses hash functions that map a point on the hypersphere into the closest vertex of the randomly rotated regular polytope so that the performance of the hashing could be improved.

The rest of the paper is organized as follows: In Sect. 2 we provide the overview of LSH algorithm. Section 3 describes the framework of Spherical LSH (SLSH). Section 4 and Sect. 5 presents the asymptotic and practical evaluation of SLSH respectively. Finally Sect. 6 concludes the paper.

2. Locality-Sensitive Hashing (LSH)

Since our proposal is a variant of the LSH algorithm, we must describe LSH. The following is a brief introduction to the LSH algorithm. A more detailed description will be found in [2]–[5].

LSH is a randomized algorithm for solving the (R, c)-NN problem. The (R, c)-NN problem is a decision version of the approximate nearest neighbor problem. It is known that the c-approximate nearest neighbor problem can be reduced to (R, c)-NN problem with complexity $O(\log(n/\varepsilon))$. In the following, c represents an approximation factor (let $c = 1 + \varepsilon$), and $\mathrm{dist}(u, v)$ represents the distance between two vectors u and v.

Definition 1: The c-approximate nearest neighbor problem is defined as follows: Given a set P of points in a d-dimensional space $\mathbb{R}^d$, devise a data structure which for any query point $q \in \mathbb{R}^d$ finds a point $p \in P$ that satisfies for all $p' \in P$, $\mathrm{dist}(p, q) \leq c \cdot \mathrm{dist}(p', q)$.

Definition 2: The (R, c)-NN problem is defined as follows: Given a set P of points in a d-dimensional space $\mathbb{R}^d$, and a parameter $R > 0$, devise a data structure which for any query point $q \in \mathbb{R}^d$ does the following:

- if there exists a point $p \in P$ s.t. $\mathrm{dist}(p, q) \leq R$ then return YES and a point $p' \in P$ s.t. $\mathrm{dist}(p', q) \leq cR$,
- if $\mathrm{dist}(p, q) > cR$ for all $p \in P$ then return NO.

In [9], Har-Peled showed the following theorem.

Theorem 1: The c-approximate nearest neighbor problem can be reduced to a (R, c)-NN problem with complexity $O(\log(n/\varepsilon))$.

LSH can solve the (R, c)-NN problem significantly faster than other existing methods, especially in high dimensional spaces. LSH uses a family of hash functions, called locality-sensitive hash functions, where the probability of collision is much higher when two points are close to each other. To construct a suitable data structure, LSH then hashes every point in the dataset into hash tables using hash functions randomly chosen from the locality-sensitive hash function family. Finding the nearest neighbor of a query point involves applying the hash functions to the query point and enumerating the points in the dataset that appear in the corresponding buckets.

The locality-sensitive hash function family is an important constituent of the LSH algorithm. For a domain S of the point set, an LSH family is defined as follows:

Definition 3: A family $\mathcal{H} = \{h : S \rightarrow U\}$ is called (r_1, r_2, p_1, p_2)-sensitive if for any $u, v \in S$,

- if $\mathrm{dist}(u, v) \leq r_1$ then $\Pr_{\mathcal{H}}[h(u) = h(v)] \geq p_1$,
- if $\mathrm{dist}(u, v) > r_2$ then $\Pr_{\mathcal{H}}[h(u) = h(v)] \leq p_2$,

where $\Pr_{\mathcal{H}}[\ ]$ indicates the probability that the equation between the brackets holds true when h is randomly chosen from $\mathcal{H}$. In order for an LSH family to be useful, it has to satisfy the inequalities $p_1 > p_2$ and $r_1 < r_2$.

For solving the (R, c)-NN problem, LSH sets $r_1 = R$ and $r_2 = cR$, and then amplifies the difference of collision probabilities by taking direct product of hash functions, i.e.,

$$g(p) = \{\, h_1(p), h_2(p), \ldots, h_k(p) \,\}, \tag{1}$$

where h_i is a (r_1, r_2, p_1, p_2)-sensitive hash function randomly chosen from the family $\mathcal{H}$. For a query point q, LSH scans only the points that stay in the same bucket as $g(q)$. Since the process is probabilistic, it could occur that the query point and the nearest point stay away from each other. In order to reduce such false negatives, the LSH algorithm makes L hash tables, and scans the points in the union of the buckets corresponding to each of $g_1(p), g_2(p), \ldots, g_L(p)$.

From the settings above, we can obtain the following theorem:

Theorem 2: LSH can solve the (R, c)-NN problem with $O(dn + n^{1+\rho})$ space and $\tilde{O}(n^\rho)$ time, where $\rho = \frac{\log 1/p_1}{\log 1/p_2}$

Now, the remaining problem is to design the locality-sensitive hash functions. The firstly proposed LSH [2], which worked for the Hamming metric space, used a random bit extraction from unary expressions. It showed the performance to be $\rho = 1/c$. The later improvement of LSH [3] extended the target metric space to arbitrary l_s-norm space with $s \in (0, 2]$, and it also improved the index ρ by using a random projection based on s-stable distributions. The most recent improvement of LSH [4] employed "ball partitioning" instead of the former "grid partitioning" to partition the space and bounded the complexity by

$$\rho = \frac{1}{c^2} + O\left(\frac{\log\log n}{\log^{1/3} n}\right). \tag{2}$$

For the practical variant, the paper [4] also proposed to use the Leech Lattice-based partitioning, which is likely to perform better than the aforementioned "ball partitioning" due to much lower "big-Oh" constants. It uses a lattice called Leech Lattice [10], which is a very symmetric lattice embedded in a 24-dimensional space. When the dimensionality is larger than 24, dimensionality reduction is needed before using Leech Lattice.

These improvements are all concerned with the problem of: "How to partition the space well?" Leech Lattice-based partitioning, which is round and symmetric, is quite a nice partitioning method except that it can be applied only to a 24-dimensional space.

Now recall that the purpose of this paper is to solve the nearest neighbor problem on a unit hypersphere. While the examples presented above are all considering the general problem to partition the entire $\mathbb{R}^d$ space, we have to solve the special problem of partitioning the unit hypersphere embedded in $\mathbb{R}^d$. Is there any partitioning that works nicely especially for the hypersphere? Our solution will be described in the next section.

3. Spherical LSH (SLSH)

Here we propose a novel locality-sensitive hash function that performs better than the previously proposed ones. While earlier LSH families are considering arbitrary points in $\mathbb{R}^d$ space as in [2]–[4], we are considering arbitrary points on the unit $(d-1)$-sphere embedded in $\mathbb{R}^d$ space with center at the origin. In other words, all we have to do is to partition the surface of the unit hypersphere in $\mathbb{R}^d$, in contrast to the fact that the previously proposed LSH algorithm [2]–[4] had to partition the entire $\mathbb{R}^d$ space. This section describes the locality-sensitive hash functions for partitioning the surface of the unit hypersphere in high dimensions. We named this process as SLSH (Spherical LSH).

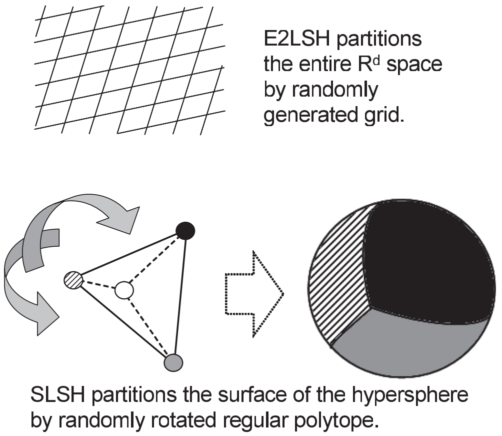
E2LSH partitions
the entire R^d space
by randomly
generated grid.
SLSH partitions the surface of the hypersphere
by randomly rotated regular polytope.

Fig. 1 The illustration of the SLSH partitioning.

3.1 Problem Description

The problem of interest in this paper is defined as follows.

Definition 4: The (R, c)-NN problem on the Unit Hypersphere: Given a set P of points in a d-dimensional space $\mathbb{R}^d$, and all points $p \in P$ satisfies that $\|p\| = 1$. Given a parameter $R > 0$, devise a data structure which for any query point $q \in \mathbb{R}^d$ satisfying $\|q\| = 1$ does the following:

- if there exists a point $p \in P$ s.t. $\text{dist}(p, q) \leq R$ then return YES and a point p' s.t. $\text{dist}(p', q) \leq cR$,
- if $\text{dist}(p, q) > cR$ for all $p \in P$ then return NO.

This is a special case of the (R, c)-NN problem described in Def. 2, having wide application area as described in Sect. 1.

3.2 Locality-Sensitive Hash Functions Using a Regular Polytope

SLSH uses a randomly rotated regular polytope for partitioning the surface of the unit hypersphere. After rotating the polytope at random, the hash function $h(p)$ is then defined as the number assigned to the vertex which is nearest to p. In other words, our hash function partitions the surface of the unit hypersphere like a Voronoi diagram. Figure 1 illustrates the concept of SLSH partitioning in 3-dimensional space.

First let us go through some notations.

Hypersphere: A hypersphere is the generalization of the sphere to higher dimensions. Often the symbol $\mathbb{S}^n$ is used to represent the n-sphere that has an n surface dimension and is embedded in an $(n+1)$-dimensional space.
A unit hypersphere is a hypersphere with a radius of unit length. From now on we will consider the unit $(d-1)$-sphere, whose center is located at the origin. Note that the $(d-1)$-sphere is embedded in $\mathbb{R}^d$.

Regular polytope: A regular polytope is the generalization of the regular polygon (in two-dimensional space) and the regular polyhedron (in three-dimensional space) to higher dimensions. It has a high degree of symmetry such as the following:

- All edges have an equal length, which means that the distance between the adjacent vertices are always the same.
- All faces are congruent.

It is known that there exist only three kinds of regular polytopes in higher ($d \geq 5$) dimensions, namely:

Simplex, having $d+1$ vertices, is analogous to the tetrahedron.
Orthoplex (Cross polytope), having $2d$ vertices, is analogous to the octahedron.
Hypercube (Measure polytope), having 2^d vertices, is analogous to the cube.

Suppose that we randomly rotate the regular polytope inscribed in a unit (d–1)-sphere. We can partition the (d–1)-sphere so that all points belong to the nearest vertex of the rotated regular polytope.

Definition 5: (The key idea of our algorithm): Let $\{\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_N\}$ ($\|\tilde{v}_i\|^2 = 1$) be a set of vertices that forms a regular polytope in $\mathbb{R}^d$ where N represents the number of vertices of the employed polytope, and let A be a rotation matrix. For an arbitrary unit vector p, a hash function $h_A(p)$ is defined as the following:

$$h_A(p) = \mathrm{argmin}_i \ \mathrm{dist}(A\tilde{v}_i, p). \qquad (3)$$

Note that for a given p, we can obtain $h_A(p)$ in $O(d^2)$ time for every type of regular polytope (it will be discussed later).

By considering A as an arbitrary rotation matrix in $\mathbb{R}^d$ space, $\mathcal{H} = \{h_A\}$ satisfies the definition of the locality-sensitive hash function family. SLSH uses this LSH family for hashing.

3.3 The Algorithm

Here we will describe the details of the algorithm. The complete algorithm is also presented in Fig. 2 and Fig. 3.

The coordinates of the vertices of the regular polytope in d-dimensional space are given by the following:

Simplex:

$$[\tilde{v}_i]_j = \delta_{ij} - \frac{d+1-\sqrt{d+1}}{d(d+1)} \qquad (i = 1, 2, \ldots, N) \qquad (4)$$

$$[\tilde{v}_{N+1}]_j = \frac{1-\sqrt{d+1}}{d} - \frac{d+1-\sqrt{d+1}}{d(d+1)} \qquad (5)$$

where $[\tilde{v}_i]_j$ represents the j-th coordinate of the i-th vertex $\tilde{v}_i$, and δ_{ij} is 1 for $i = j$ and 0 otherwise.

```
simplex — The range of the function is: $h_A(p) \in \{1, \ldots, d+1\}$
    Preprocessing:
        Let $\{\tilde{v}_i \mid i = 1, \ldots, d+1\}$ be a set of vectors described in Eq. (4) and (5).
        Let $A$ be a random rotation matrix.
        For $i \leftarrow 1$ to $d+1$ do
            $v_i \leftarrow A\tilde{v}_i$
        (Then $\{v_1, v_2, \cdots, v_{d+1}\}$ forms a randomly rotated simplex.)
    Calculation of $h_A(p)$ for input $p$:
        $h_A(p) \leftarrow \mathrm{argmax}_i(v_i \cdot p)$
        Return $h_A(p)$

orthoplex — The range of the function is: $h_A(p) \in \{1, \ldots, 2d\}$
    Preprocessing:
        Let $v_i$ be the $i$-th column of a random rotation matrix $A$.
        (Then $\{v_1, v_2, \cdots, v_d, -v_1, -v_2, \cdots, -v_d\}$ forms a randomly rotated orthoplex.)
    Calculation of $h_A(p)$ for input $p$:
        $h_A(p) \leftarrow \mathrm{argmax}_i |v_i \cdot p|$.
        if $(v_{h_A(p)} \cdot p) < 0$ then $h_A(p) \leftarrow h_A(p) + d$
        Return $h_A(p)$

hypercube — The range of the function is: $h_A(p) \in \{0, \ldots, 2^d - 1\}$
    Preprocessing:
        Let $v_i$ be the $i$-th column of a random rotation matrix $A$.
        (Then $\{u_{s_1 s_2 \cdots s_d} = \sum s_i v_i \mid s_i = \pm 1\}$ forms a randomly rotated hypercube.)
    Calculation of $h_A(p)$ for input $p$:
        $h_A(p) \leftarrow 0$
        For $i \leftarrow 1$ to $d$ do
            if $(v_i \cdot p) \geq 0$ then $h_A(p) \leftarrow h_A(p) + 2^{i-1}$
        Return $h_A(p)$
```

Fig. 2 The implementation of $h_A(p)$ for each type of regular polytope.

```
How to make a random rotation matrix:
  For $i \leftarrow 1$ to $d$ do
      Let $v_i$ be a random vector from the $d$-dimensional Gaussian distribution.
      For $j \leftarrow 1$ to $i-1$ do
          $v_i \leftarrow v_i - (v_i \cdot v_j) v_j / |v_j|$
              (Gram-Schmidt orthogonalization)
      $v_i \leftarrow v_i / |v_i|$
          (normalize to unit length)
  Return $(v_1 \ v_2 \ \cdots \ v_d)$
```

Fig. 3 Algorithm for making a random rotation matrix.

Orthoplex: All permutations of $(\pm 1, 0, 0, \cdots, 0)$ give the coordinates of the vertices. It follows that orthoplex has $2d$ vertices.
Hypercube: $\frac{1}{\sqrt{d}} \times (\pm 1, \pm 1, \cdots, \pm 1)$ give the coordinates of the vertices. It follows that the hypercube has 2^d vertices.

Let us consider how to obtain the nearest vertex efficiently. Instead of directly solving Eq. (3), it is computationally easier to solve

$$h_A(p) = \mathrm{argmax}_i(A\tilde{v}_i \cdot p). \qquad (6)$$

If we calculate $\{v_i = A\tilde{v}_i \mid i = 1, \cdots, N\}$ in advance, a $d + 1$ dot-product calculation would suffice to return $h_A(p)$ for the simplex. For the orthoplex, doing a $2d$ dot-product calculation is also possible; however, a more efficient way exists. If v is a vertex of the orthoplex, then $-v$ is also a vertex of the orthoplex. The dot-product of $-v$ and p is just $-(v \cdot p)$. Therefore, we do not have to calculate the dot-product $2d$ times — a calculation of only d times will suffice. For hypercube, naively solving Eq. (6) needs a 2^d times dot-product calculation that is of course infeasible. However, a way exists to avoid such prohibitive calculations. The same partitioning can be obtained by a d times bisection using orthonormal basis vectors $\{e_1, \cdots, e_d\}$. We can bisect the hypersphere using each one of the basis vectors, i.e., $Bisec_{e_i}(p) = 1$ if $(e_i \cdot p) \geq 0$ and $Bisec_{e_i}(p) = 0$ otherwise. Then we could construct the map $p \in \mathbb{S}^{d-1} \mapsto \{0, 1\}^d \mapsto \{0, 1, \cdots, 2^d - 1\}$. This partitioning is equivalent to a partitioning based on the nearest vertices of a hypercube. Thus, we can conclude the following: for every type of regular polytope, $h_A(p)$ can be calculated in $O(d^2)$ time.

Let us discuss the preprocessing cost. Preprocessing of SLSH costs $O(d^3 + d^2 n)$ time for one hash function. The former d^3 is the cost to make random rotation matrix. The latter $d^2 n$ is to hash all points in the dataset. The memory space overhead is $O(d^2 L)$ to store the rotated vertices, and $O(nL)$ to store the hash index of all points in the datasets. Note that the number of the hash tables L can be bounded by n^ρ.

4. Computational Analysis of the Exponent ρ

As mentioned before, the performance of LSH is evaluated by the index $\rho = \frac{\log 1/p_1}{\log 1/p_2}$. In this section we describe the evaluation of the ρ.

Again, p_1 is the probability of collision of two points with a distance of R, and p_2 is the probability of collision of two points with a distance of cR. In the original LSH they can change the scale of the coordinate; thus, they can assume $R = 1$ without loss of generality. On the other hand, we cannot scale the coordinate because SLSH works only to $\mathbb{S}^{d-1}$. Therefore the performance index ρ of SLSH must be evaluated for several Rs.

For ease of comparison, we evaluated several types of locality-sensitive hash functions. The candidates were as follows:

SLSH (Our proposal): Partitioning based on the rotated regular polytope.

Leech Lattice: Proposed in [4], with the dimensionality reduction of d to 24.

Spherical Bisection: Partitioning also referred to as random hyperplane-based hash functions [11]. In contrast to our "hypercube-partitioning," which bisects the hypersphere with an *orthonormal* set of vectors, the Spherical Bisection method bisects the hypersphere with a *random* set of vectors.

Let $p(r)$ represent the collision probability of a single hash function with respect to the distance r, i.e., $p(r) = \Pr_{\mathcal{H}}[h(u) = h(v)]$ for the two points u and v that satisfy $\mathrm{dist}(u, v) = r$.

In Table 1, the values of $p(r)$ are displayed. The values for the Leech Lattice were cited from [4]. The values for the Spherical Bisection were analytically obtained by the equation $p(r) = 1 - \theta/\pi$, where θ represents the angle between two vectors measured in radians. The cosine of θ and the Euclidean distance r has the relationship as $r^2 = 2(1 - \cos\theta)$. The values $p(r)$ of the SLSH were computed by the Monte-Carlo simulation for 10^6 trials. Note that the value $p(r)$ does not depend on input sets, distributions, or anything like that: it relies on the probability of two points being hashed to the same value, which can be computed using Monte-Carlo methods fairly accurately.

Figures 4 and 5 plot the value of ρ vs. $R = r_1$. As mentioned before, the performance of the Leech Lattice method does not depend on R because the coordinate scale is changeable in the Leech Lattice method. Therefore, only the best ρ over R is plotted for the Leech Lattice. For the SLSH and the Spherical Bisection, the value of $\rho = \frac{\log 1/p_1}{\log 1/p_2}$ was calculated from $p(r)$ displayed in Table 1. In the calculation we did not use the value $p(r)$ less than 0.00001 because such values are less reliable and inappropriate for SLSH implementation.

From these figures, we can observe the following: For almost all dimensionalities and polytopes, SLSH performs better than the Spherical Bisection method. Comparing ρ on the same polytopes, larger dimensionality implies better ρ. It means that our method could avoid the curse of dimensionality. Comparing ρ on various types of polytopes, simplex and orthoplex tend to show a similar result except that the orthoplex shows slightly better result when R becomes larger. The hypercube shows quite different behavior from the others. The collision probability $p(r)$ of the hypercube rapidly drops to near zero, especially in high dimensional spaces. Although the index ρ is lower than the others are, its use for a practical application is difficult because too small $p(r)$ prevent the proper adjustment of k and L, that are restricted to positive integer (this fact will be discussed again in the next section). Comparing SLSH with the Leech Lattice-based method, in the case of $c = 1.5$, for almost all dimensionalities, SLSH performs better than the Leech Lattice-based method when R is larger than 0.60–0.64. In the case of $c = 2.0$, for almost all dimensionalities, SLSH performs better than the Leech Lattice-based method when R is larger than 0.48–0.52. We displayed some of the representative values in Table 2.

This result implies that in implementing the c-approximate nearest neighbor problem solver, where we have to employ the (R, c)-NN solver hierarchically as $R = C_0 c^\lambda$ for $\lambda = 1, 2, \cdots, \Lambda$, SLSH can replace the Leech Lattice-based LSH at least some hierarchies.

Table 1 Probabilities of collision for two points with the distance of r. The values for the Leech Lattice are cited from [4]. The values for the spherical bisection are obtained analytically. The other values displayed below are obtained through our Monte-Carlo simulation for 10^6 trials.

Distance r	LeechLattice ($d > 24$)	Spherical bisection	16-dim simplex	16-dim orthoplex	16-dim hypercube	64-dim simplex	64-dim orthoplex	64-dim hypercube
0.10		0.96815	0.90133	0.88612	0.59084	0.87169	0.85846	0.12152
0.20		0.93623	0.80746	0.77939	0.33587	0.75356	0.73061	0.01271
0.30		0.90414	0.71800	0.67894	0.18092	0.64590	0.61412	0.00116
0.40		0.87181	0.63309	0.58535	0.09186	0.54531	0.50879	0.00008
0.50		0.83913	0.55276	0.49754	0.04315	0.45407	0.41365	0.00000
0.60		0.80602	0.47649	0.41595	0.01836	0.37078	0.32937	0.00000
0.70	0.08535	0.77236	0.40459	0.34066	0.00676	0.29652	0.25503	0.00000
0.80	0.05259	0.73802	0.33750	0.27211	0.00212	0.23071	0.19144	0.00000
0.90	0.03117	0.70284	0.27473	0.21051	0.00050	0.17326	0.13748	0.00000
1.00	0.01779	0.66666	0.21676	0.15533	0.00006	0.12449	0.09314	0.00000
1.10	0.00975	0.62925	0.16419	0.10797	0.00000	0.08456	0.05854	0.00000
1.20	0.00515	0.59033	0.11785	0.06906	0.00000	0.05260	0.03326	0.00000
1.30	0.00266	0.54953	0.07826	0.03872	0.00000	0.02921	0.01656	0.00000
1.40	0.00133	0.50636	0.04622	0.01789	0.00000	0.01378	0.00644	0.00000
1.50	0.00067	0.46010	0.02253	0.00587	0.00000	0.00504	0.00181	0.00000
1.60	0.00033	0.40966	0.00775	0.00108	0.00000	0.00117	0.00028	0.00000
1.70	0.00016	0.35320	0.00124	0.00006	0.00000	0.00010	0.00001	0.00000
1.80	0.00008	0.28713	0.00003	0.00000	0.00000	0.00000	0.00000	0.00000
1.90	0.00004	0.20216	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
2.00	0.00002	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

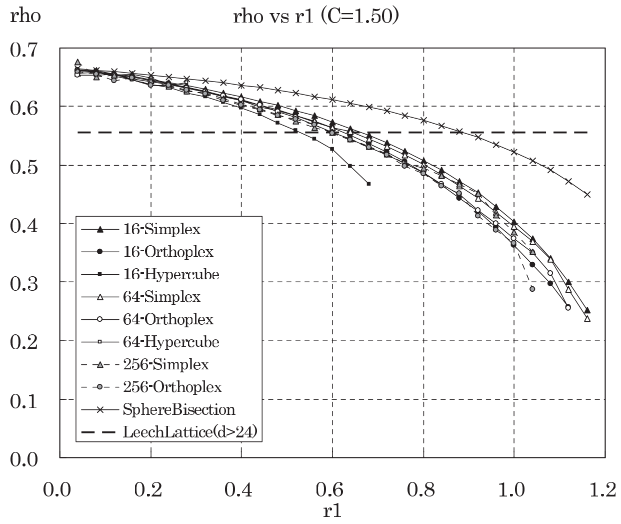

Fig. 4 The values of $\rho = \frac{\log p_1}{\log p_2}$ for $c = 1.5$.

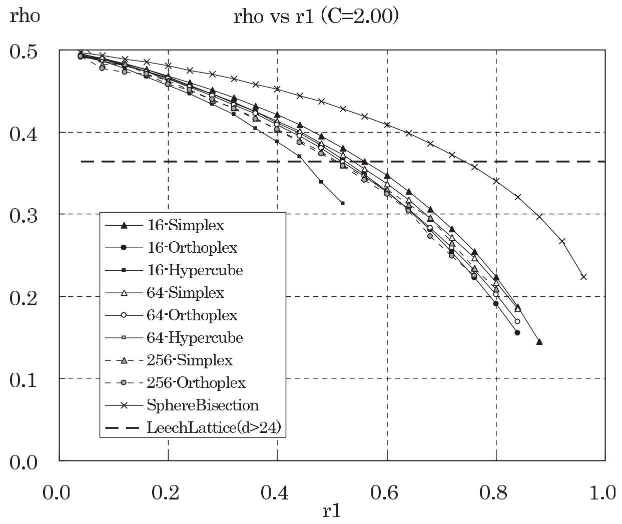

Fig. 5 The values of $\rho = \frac{\log p_1}{\log p_2}$ for $c = 2.0$.

Table 2 The values of ρ.

R	c	Leech Lattice ($d > 24$)	SLSH for 64-dim orthoplex
0.64	1.5	0.5563	0.5471
0.72	1.5	0.5563	0.5189
0.80	1.5	0.5563	0.4858
0.56	2.0	0.3641	0.3456
0.64	2.0	0.3641	0.3063

4.1 Discussion

Why could SLSH outperform the original LSH? One of the reasons is that SLSH handles the points directly in the original dimension, i.e., it does not need the dimensionality reduction. In dimensionality reduction, one cannot avoid the chance of far away points colliding to the near point; that is the disadvantage of dimensionality reduction. We can avoid such disadvantages by not using the dimensionality reduction. We have devised a good partitioning method that can be applied to any dimensional hypersphere. In our partitioning, the point $\boldsymbol{p_1}$ and $\boldsymbol{p_2} = -\boldsymbol{p_1} + \boldsymbol{\varepsilon}$, where $\boldsymbol{\varepsilon}$ is an arbitrary small vector, will *never* collide for any type of polytope, say, simplex, orthoplex, or hypercube. If we try to guarantee that $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ will never collide by the Spherical Bisection method, we need at least d times partitioning (It is similar to our hypercube method). However, like our hypercube method, it tends to partition the space too thin. It may be understood by the fact that it partitions the space into

2^d fragments. On the other hand, our simplex method and orthoplex method partition the space into $d + 1$ or $2d$ fragments. It is a milder partitioning than that of the partitioning into 2^d fragments, but points far away will collide with very little probability.

5. Performance Evaluation in the Practical Experiment

In the previous section we have made the computational analysis to evaluate the exponent ρ of the complexity $O(n^\rho)$. It was the asymptotic evaluation as the population n tends to infinity. On the other hand, this section will produce the practical evaluation of the computational cost with the moderate population n.

5.1 The Problem to Be Solved

In the following experiment, the algorithms were assigned to retrieve *all* points in the dataset whose distance to the query point is less than $R = 0.8$. This problem, called reporting problem, is a bit different from the problem stated in Def. 4. In the reporting problem we cannot bind the running time because the output could be large if a large fraction of the data point are located close to the query point. However, in usual situations this reporting problem behaves similarly to the (R, c)-NN problem and regarded to be appropriate to evaluate the (R, c)-NN algorithm [5].

5.2 Candidate Algorithms

Here we compare SLSH with E2LSH and Brute-Force algorithm. E2LSH package [12] is a latest published implementation of LSH released by the original author. E2LSH depends basically on the algorithm described in [3], with some practical improvements applied. One of such improvements is, for instance, E2LSH can reduce the number of hash function computation from kL into km (where $m \sim O(\sqrt{L})$) by using pseudo-independent functions instead of using truely independent functions.

5.3 Evaluation Criteria

The computational cost of LSH is mainly consists of two parts: hash function computation and distance computation. We evaluated these costs by counting the number of basic operations.

The cost of hash function computation is obtained by multiplying a cost for computing a hash function by the number of hash functions. In E2LSH, the operation to compute a hash function is an inner product computation of the query point and randomly generated point. We assumed this cost to be $2d$ basic operations. Since E2LSH has to compute the inner product km times per query, the total cost for hashing is $2dkm$ basic operations per query. In SLSH, we have to compute d inner products (or $d + 1$ inner products for the case of SLSH-simplex) for computing each hash function. Since we have to compute kL hash functions per query, the resulting total cost is $2d^2kL$ basic operations (or $2d(d+1)kL$ basic operations for SLSH-simplex) per query.

The cost of distance computation is obtained by multiplying a cost of distance computation for a pair of points by the number of points found in the buckets. The cost for computing a distance is assumed to be $3d$. The numbers of points found in the buckets were obtained in the experiment.

5.4 Datasets and Settings

In our experiment, the algorithms were assigned to retrieve all points that have distance less than $R = 0.8$ from the query point.

Both query points and dataset points are generated synthetically. The query points were 100 randomly generated points. The generation of the dataset points was a bit more complicated. The distribution of the distance between two randomly generated points on the hypersphere is sharply peaked at $\sqrt{2}$ when the dimensionality is high. In such cases, the random method rarely generate the point within distance 0.8 from the query points: in fact, the probability that two randomly generated points are staying within distance 0.8 is about 10^{-3} in $\mathbb{R}^{16}$ space, and about 10^{-5} in $\mathbb{R}^{32}$ space. If we made all the points completely at random, the ground truth, the points to be retrieved, was likely to be an empty set, that would make the experiment meaningless. To avoid such problem, we planted a point for each query point instead of generating all the points completely at random. That is, we intentionally made a point that had distance almost equal but slightly less than 0.8 from the query point. In this way each query point was guaranteed to have at least one point within distance 0.8 from them. Except a planted point, all other points in the dataset were generated at random. The size of the dataset was varied from 5×10^2 to 5×10^5. The dimensionalities we have tested were 8, 16, 32, 64 and 100. Throughout all experiments, we set the expected probability for each point successfully retrieved to be 0.9, i.e., $\delta = 0.1$.

To use LSH, we need to specify the parameters k and L. The parameter L was set to minimum integer that satisfies $L \geq \log \delta / \log(1 - p_1^k)$, where p_1 is the probability of two points with distance $R = 0.8$ hashed to the same buckets, its value can be found in the Table 1 in the previous section. To determine the parameter k, we had two choices. One choice was to set k as $\log_{1/p_2} n$, which guarantees the minimum computational cost when $n \to \infty$. However, such a k does not necessarily give the minimum computational cost for the real dataset where $n < \infty$. Therefore, we took another choice; that is, we experimentally estimated the computational cost for several ks by sampling method and chose the k that gives the minimum computational cost. E2LSH already has the function to do this process automatically. We used the same process for SLSH, too. Note that there exists another necessary condition on k and L. Since the LSH data structure requires $O(nL)$ memory space, there exists upper bound for L depending on the available physical mem-

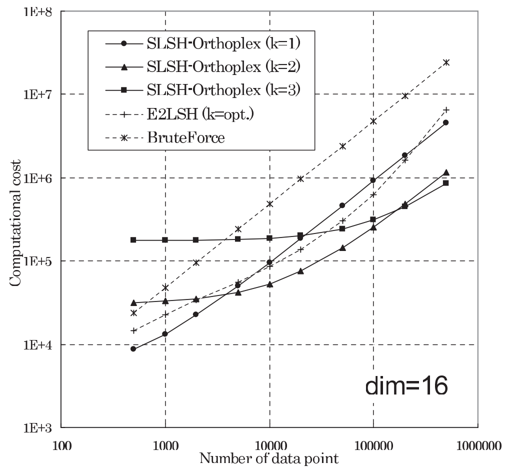

Fig. 6 The computational cost of SLSH-orthoplex, E2LSH and Brute-Force search with dimensionality d = 16.

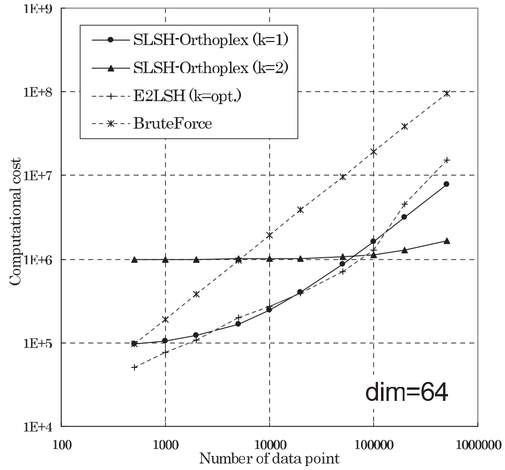

Fig. 7 The computational cost of SLSH-orthoplex, E2LSH and Brute-Force search with dimensionality d = 64.

ory. In choosing the parameters, we took this condition into consideration. A more detailed inspection of the amount of memory requirement will be carried out in Sect. 5.6.

5.5 Results

In our experiments, recall ratio, the ratio of the number of actually retrieved points to the number of ground truth, were always in the range of 85–100%; that means all of the nominated algorithms have worked correctly.

The computational cost of SLSH with each type of polytopes will be described severally in the following.

5.5.1 SLSH with Orthoplex

The computational cost of SLSH with orthoplex is plotted in Figs. 6 and 7, together with the cost of E2LSH and Brute-Force method. In these figures we dare to plot the multiple curves for SLSH with several ks instead of single curve for optimal k, in order to make the effect of k observable. Here we displayed only d = 16 case and d = 64 case. For other dimensionalities, the results for d = 8 and d = 32 were similar to Fig. 6, and the result for d = 100 was similar to Fig. 7.

Table 3 The values of k and L. (L is determined by k).

	16 dim.		64 dim.	
	k	L	k	L
E2LSH	20	595	20	595
E2LSH	24	1485	24	1485
E2LSH	28	3741	28	3741
SLSH Simplex	1	6	1	9
SLSH Simplex	2	20	2	42
SLSH Simplex	3	59	3	186
SLSH Simplex	4	177	4	811
SLSH Orthoplex	1	8	1	11
SLSH Orthoplex	2	30	2	62
SLSH Orthoplex	3	114	3	322
SLSH Orthoplex	4	419	4	1677
SLSH Hypercube	1	1085	-	-
SLSH Hypercube	2	512805	-	-

In Fig. 6, we can observe that the optimal k for SLSH increases as the number of data points increases. Note that the optimal k for E2LSH also increases as the number of data points increases (even though that is not displayed in the figure), and that the increase of k leads to the increase of L (Table 3).

We can also observe that SLSH showed highest performance at any n while parameter k was properly set up. Note that the improvement over Brute-Force method has increased as the number of the points increases.

It is worth pointing out that the performance of E2LSH have gotten worse when $n \geq 5 \times 10^4$. That was because the available memory restriction prevented E2LSH to use the best parameters k and L (Recall that the amount of available memory determines the upper bound for L). For example, for the settings $d = 16$ and $n = 1 \times 10^5$, the maximum L allowed was around 1600 when we used a computer with 2.00 GB memory. Even though E2LSH had estimated the optimal parameters as $(k, L) = (28, 3741)$, E2LSH had to use the alternative parameters within the memory restriction as a compromise, that caused the degradation of the computational cost. On the other hand, our algorithm did not suffer from the memory restriction as far as $n \leq 5 \times 10^5$. The advantage of our algorithm comes from the fact that the optimal parameters k and L of the SLSH is smaller than E2LSH. For example, the estimated optimal parameters for SLSH in the same settings described above were $(k, L) = (2, 30)$. The further discussion of memory restriction will be resumed in Sect. 5.6.

In higher dimensional case, shown in Fig. 7, the advantage of SLSH decreased especially with low populations. That is because the hashing cost of SLSH is $O(d^2)$ while that of E2LSH is only $O(d)$. However, since SLSH could reduce the number of distance calculation more efficiently, SLSH could display its advantage as n increases. Again the performance of E2LSH have gotten worse when $n \geq 5 \times 10^4$ because of the lack of memory. On the other hand, our method

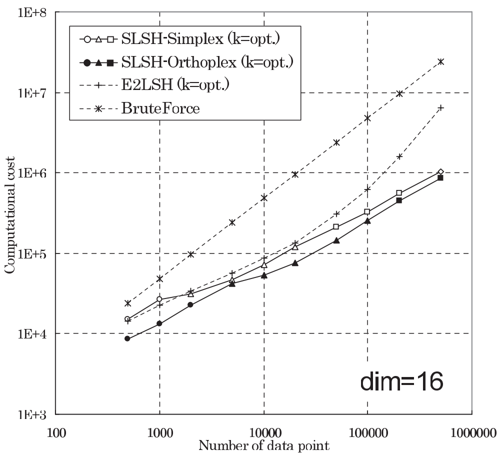

Fig. 8 The computational cost of SLSH-simplex and SLSH-orthoplex with the dimensionality $d = 16$. In the figure, the circle represents the parameter $k = 1$, the triangle represents the parameter $k = 2$, and the square represents the parameter $k = 3$.

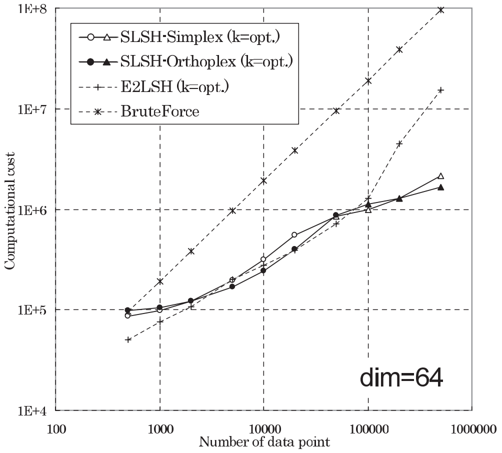

Fig. 9 The computational cost of SLSH-simplex and SLSH-orthoplex with the dimensionality $d = 64$. In the figure, the circle represents the parameter $k = 1$ and the triangle represents the parameter $k = 2$.

worked well as far as $n \leq 5 \times 10^5$.

5.5.2 SLSH with Simplex

SLSH with simplex showed similar result as SLSH with orthoplex, with the latter slightly better (Figs. 8 and 9). This result is understood by the fact that the SLSH-orthoplex has slight advantage both in index ρ and the number of inner product calculation per query.

5.5.3 SLSH with Hypercube

Even though the SLSH with Hypercube could produce the lowest ρ in the asymptotic computational cost $O(n^\rho)$, the virtue seemed to be diminished until n increases significantly (Fig. 10). As seen in Table 3 with $d = 16$ case, the very low $p(r)$ of hypercube (see Table 1) causes quite large L even we choose the minimum k setting: $k = 1$. Furthermore, with $d = 64$ case we cannot even calculate optimal L because $p(r)$ was too small to be estimated by the Monte-Carlo method. We can guess that the optimal L would be so large that the implementation will be prohibited by the restriction of memory. By these reason, SLSH-hypercube would be useful only when the dimensionality is moderate and the size of the dataset is quite large.

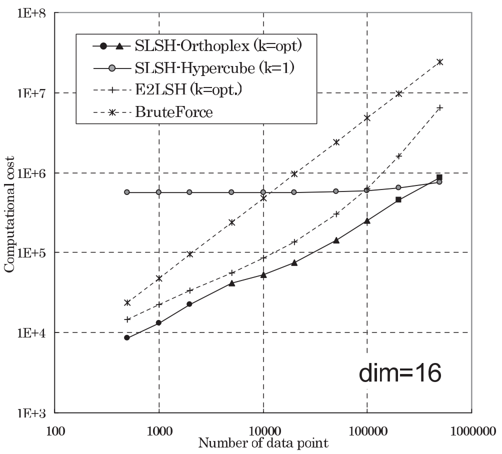

Fig. 10 The computational cost of SLSH-hypercube and SLSH-orthoplex with the dimensionality $d = 16$. In the figure, the circle represents the parameter $k = 1$, the triangle represents the parameter $k = 2$, and the square represents the parameter $k = 3$.

5.6 Memory Requirment

Here we resume the discussion of the memory requirement, mainly with respect to the dimensionality of the data. The amount of memory requirement for LSH (both E2LSH and SLSH) is the total of three main components, namely, the space to store the data itself, the space to store the hash functions and the space to store the hash tables (referred to as LSH structure). We have calculated the memory requirement for various data dimensionalities, where the population n is fixed to 1×10^5. The results are summarized in Figs. 11 and 12. Note that in this calculation we calculated the optimal parameters k and L *without* considering the memory restriction, as opposed to the experiment in Sect. 5.5.

As shown in Fig. 11, the memory requirement for E2LSH is always over 2.00 GB. It is consistent with the fact described in Sect. 5.5, i.e., the performance of E2LSH have gotten worse at $n \geq 5 \times 10^4$ because of the lack of memory. The most part of this memory requirement comes from the LSH structure. In LSH, we have to maintain L hash tables and each of the tables stores n points. The estimated optimal L is large as 3741 for all dimensionalities, that results in this huge memory requirement for LSH structure.

On the other hand, Fig. 12 indicates that SLSH needs less memory for LSH structure (The reason why LSH structure for 128 dimension requires less memory than 64 dimension is that the optimal k changes from two to one between these two settings, and it brings the smaller optimal L for

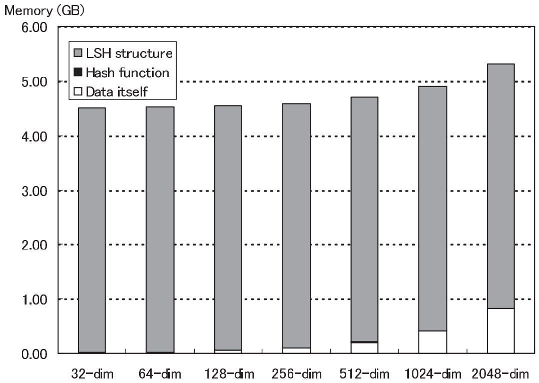

Fig. 11 The memory requirement for E2LSH ($n = 1 \times 10^5$).

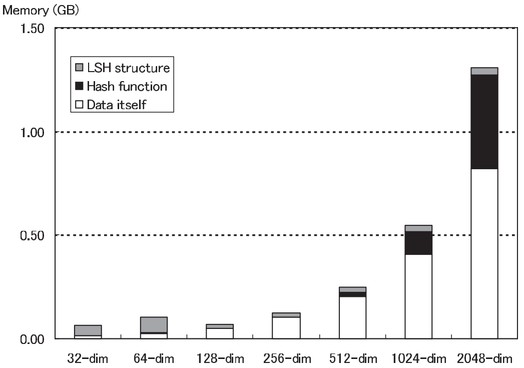

Fig. 12 The memory requirement for SLSH-orthoplex ($n = 1 \times 10^5$).

128-dimension setting). Considering the fact that this memory requirement is sensitive to the size of the dataset — it is proportional to $nL = O(n^{1+\rho})$ — we may say that SLSH is memory-efficient as far as regarding with the size of the dataset.

Meanwhile, we have to mention the drawback of SLSH. In SLSH we need $O(d^2kL)$ memory to store the hash functions, while E2LSH requires only $O(dk\sqrt{L})$. Even though the k and L for the SLSH are smaller than E2LSH, the significance of the term d^2 increases as the dimensionality increases, as seen in Fig. 12. From the figure, we may presume that SLSH is not applicable when the dimensionality increases to tens of thousands. Nevertheless, we may still say that SLSH is memory-efficient regarding with the size of the dataset because the increase of this part of memory requirement is less sensitive to the size of the dataset: it is proportional to $L = O(n^{\rho})$. Considering these facts, our future work should include a reduction of the memory space to store the hash functions. The fact that E2LSH reduces it from $O(dkL)$ into $O(dk\sqrt{L})$ by using non-independent hash functions might be a good precedent for our future work.

6. Conclusion

In this paper we have proposed an algorithm to solve the approximate nearest neighbor problem when all points are constrained to lie on the surface of the unit hypersphere. Our algorithm, named SLSH, is based on the LSH scheme, and outperforms state-of-the-art LSH variants. The computational analysis of the exponent ρ described in Sect. 4 has shown the asymptotic improvement of SLSH, that guarantees SLSH outperforms the previously proposed LSH when $n \to \infty$. The practical experiment described in Sect. 5 compared the three types of SLSH and E2LSH. Since SLSH-orthoplex has displayed its advantage in both computational cost and memory requirement compared with E2LSH, it is the best solution for the approximate nearest neighbor search for a dataset of normalized vectors, unless the dimensionality is as large as tens of thousands. SLSH-hypercube might be another good solution when the size of the dataset is considerably large and the dimensionality of the space is not so large.

Acknowledgements

We would like to thank Mr. Alexandr Andoni and Prof. Piotr Indyk for providing the E2LSH package. This work was supported in part by Hokkaido University Global COE Program "Center for Next-Generation Information Technology based on Knowledge Discovery and Knowledge Federation" of MEXT (Ministry of Education, Culture, Sports, Science and Technology), Grant-in-Aid for Young Scientists (B) No. 19700079 of MEXT, and PRESTO of JST (Japan Science and Technology Agency).

References

[1] J.L. Bentley, "Multidimensional binary search trees used for associative searching," Commun. ACM, vol.18, no.9, pp.509–517, 1975.
[2] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," Proc. 25th International Conference on Very Large Data Bases, VLDB1999, pp.518–529, 1999.
[3] M. Datar, P. Indyk, N. Immorlica, and V. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," Proc. Symposium on Computational Geometry 2004, pp.253–262, 2004.
[4] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," Proc. 47th Annual IEEE Symposium on Foundations of Computer Science, FOCS'06, pp.459–468, 2006.
[5] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," Commun. ACM, vol.51, no.1, pp.117–122, 2008.
[6] D.G. Lowe, "Object recognition from local scale-invariant features," Proc. 7th International Conference on Computer Vision, ICCV'99, vol.2, pp.1150–1157, 1999.
[7] D.G. Lowe, "Distinctive image features from scale-invariant keypoints," Int. J. Comput. Vis., vol.60, no.2, pp.91–110, 2004.
[8] G. Salton and M.J. McGill, Introduction to modern information retrieval, McGraw Hill, 1983.
[9] S. Har-Peled, "A replacement for voronoi diagrams of near linear size," Proc. 42nd Annual Symposium on Foundations of Computer Science, FOCS'01, pp.94–103, 2001.
[10] J. Leech, "Notes on sphere packings," Canadian Journal of Mathematics, vol.19, pp.251–267, 1967.
[11] M.S. Charikar, "Similarity estimation techniques from rounding algorithms," Proc. 34th Annual ACM Symposium on Theory of Computing, STOC'02, pp.380–388, 2002.
[12] A. Andoni and P. Indyk, "E2LSH: Exact Euclidean locality-sensitive hashing," http://web.mit.edu/andoni/www/LSH/

Kengo Terasawa received B.E. and M.E. degrees in civil engineering from the University of Tokyo in 1998 and 2000 respectively. He received Ph.D. in systems information science from Future University-Hakodate in March 2006. He was a postdoctral researcher at Venture Business Laboratory, Hokkaido University during 2006–2007; and a Global COE postdoctral researcher at Hokkaido University during 2007–2008. He is now a research associate at Future University-Hakodate. Since 2008, he has also been a researcher of PRESTO, Japan Science and Technology Agency (JST). His current research interest includes image processing, information retrieval and matching algorithm.

Yuzuru Tanaka received his B.E. and M.E. degrees in information engineering from Kyoto University in 1972 and 1974 respectively. He received Dr. Eng. degree in information engineering from the University of Tokyo in July 1985. He was a research associate at Hokkaido University during 1974–1977, a lecturer during 1977–1985, and an associate professor during 1985–1990. Since 1990, he has been a professor at Hokkaido University. He has also been a Director of Meme Media Laboratory at Hokkaido University since 1996, and a visiting professor at National Institute of Informatics since 2004. His research area involves database and media architectures and knowledge federation architectures. He recieved Nikkei Business Publications Grand Prize of Technological Achievement in 1994 for the R&D on 2D meme media architecture IntelligentPad. He is a member of IPSJ, JSSST, JSAI and IEEE.