Improved locality-sensitive hashing method for the approximate nearest neighbor problem

Lu Ying-Hua(陆颖华)[a], Ma Ting-Huai(马廷淮)[a)b)†], Zhong Shui-Ming(钟水明)[a], Cao Jie(曹 杰)[c], Wang Xin(王 新)[d], and Abdullah Al-Dhelaan[e]

[a] *Jiangsu Engineering Center of Network Monitoring, Nanjing University of Information Science & Technology, Nanjing* 210044, *China*
[b] *School of Computer & Software, Nanjing University of Information Science & Technology, Nanjing* 210044, *China*
[c] *School of Economic & Management, Nanjing University of Information Science & Technology, Nanjing* 210044, *China*
[d] *Huafeng Meteorological Media Group, Beijing* 100086, *China*
[e] *Computer Science Department, College of Computer and Information Science, King Saud University, Riyadh* 11362, *Saudi Arabia*

(Received 16 October 2013; revised manuscript received 10 February 2014; published online 10 June 2014)

In recent years, the nearest neighbor search (NNS) problem has been widely used in various interesting applications. Locality-sensitive hashing (LSH), a popular algorithm for the approximate nearest neighbor problem, is proved to be an efficient method to solve the NNS problem in the high-dimensional and large-scale databases. Based on the scheme of *p*-stable LSH, this paper introduces a novel improvement algorithm called randomness-based locality-sensitive hashing (RLSH) based on *p*-stable LSH. Our proposed algorithm modifies the query strategy that it randomly selects a certain hash table to project the query point instead of mapping the query point into all hash tables in the period of the nearest neighbor query and reconstructs the candidate points for finding the nearest neighbors. This improvement strategy ensures that RLSH spends less time searching for the nearest neighbors than the *p*-stable LSH algorithm to keep a high recall. Besides, this strategy is proved to promote the diversity of the candidate points even with fewer hash tables. Experiments are executed on the synthetic dataset and open dataset. The results show that our method can cost less time consumption and less space requirements than the *p*-stable LSH while balancing the same recall.

Keywords: approximate nearest neighbor problem, locality-sensitive hashing

PACS: 02.50.Cw, 02.50.Ey, 02.50.Fz, 02.60.Gf **DOI:** 10.1088/1674-1056/23/8/080203

1. Introduction

The nearest neighbor search (NNS) problem seems like a simple problem: given a dataset with n points $P = \{p_1, \ldots, p_n\}$ in a metric space X with the distance function d, preprocess P so as to efficiently answer queries for finding the point in P closest to a query point $q \in X$, and then returns the identity of the closest point. This is an important issue in a variety of interesting domains, for instance, image retrieval, data mining, data compression, privacy preserving, machine learning, pattern recognition, statistics and data analysis, and so on. An ideal indexing scheme for the NNS problem should have the following properties: accuracy, time efficiency, space efficiency, and highly dimensional.[1] Furthermore, the construction of the indexing data structure should be rapid and it should be convenient to handle the various sequences of insertions and deletions. At present, there are two thoroughly researched and distinct approaches to the NNS problem: the deterministic search method and the probabilistic search method.

Deterministic search methods are usually implemented on the basis of data structures using spatial partitions and recursive hyperplane decomposition and tree-based search structures, such as the kd-tree,[2] R-tree,[3] and the Voronoi diagram.[4] Probabilistic search methods do not guarantee an exact result but instead of providing a high probability guarantee for obtaining the correct outcome or one close to it, have been proved to be effective solutions to allow fast queries even for high-dimensional datasets while maintaining good precision. Several approximate search algorithms have been brought forward, such as K-means[5] and the NV Tree.[6] Locality-sensitive hashing (LSH)[7,8] is one of the most popular techniques to find the nearest neighbor from a dataset in an approximate manner, which will be described in detail in Section 2. The principle of LSH is to use multiple hash functions to project similar data to the same bucket with a higher probability via quantization rather than dissimilar points, i.e., if two points are close together initially, after a "projection" operation these two points will still be close together. In the nearest neighbor search for a query, LSH efficiently reduces the search range to the points stored in the same hash buckets as the query point. By means of comparing the similarity between these points according to a certain metric, LSH successfully decreases the computational overhead with a complexity sublinear in the amount of data.

*Project supported by the National Natural Science Foundation of China (Grant No. 61173143) and the Special Public Sector Research Program of China (Grant No. GYHY201206030). The authors extend their appreciation to the Deanship of Scientific Research at King Saud University for funding this work through research group No. RGP-VPP-264.

†Corresponding author. E-mail: thma@nuist.edu.cn

© 2014 Chinese Physical Society and IOP Publishing Ltd

In the different kinds of variations of LSH, one of the most popular algorithms is the p-stable LSH proposed by Datar *et al.*[9] The authors present a novel locality-sensitive hashing scheme for the approximate nearest neighbor problem under l_p norm, based on p-stable distributions. They proved that their algorithm can find the exact near neighbor in $O(\log n)$ time for data satisfying a certain "bounded growth" condition. The proposed p-stable LSH can directly work on points in the Euclidean space without any embedding, which is free of large factors and is easy to implement.

In spite of the above advantages of the p-stable LSH, it still has some important restrictions. Like the basic LSH algorithm, p-stable LSH needs large memory consumption for storing the many hash tables to maintain both recall and high precision. Besides, its speed of finding the nearest neighbor can also be improved to some extent. In this paper, we present an overview of the traditional p-stable LSH and propose a new LSH algorithm called randomness-based locality-sensitive hashing (RLSH) based on the p-stable LSH. The RLSH does not need the same number of hash tables as the number of hash tables which the p-stable LSH needs to keep the same precision and recall as the p-stable LSH. The query point just needs to randomly select a certain hash table to project. Points that collide with those points, which collide with the query point in other hash tables, compose the candidate dataset including those points. The RLSH can improve space efficiency and search quality with less query time. Through experiments, we can verify the superiority of our algorithm by comparing it with the traditional p-stable LSH.

The rest of this paper is organized as follows. Section 2 presents an overview of locality-sensitive hashing. In Section 3, we introduce the basic p-stable locality-sensitive hashing algorithm. Section 4 describes our proposed algorithm RLSH. Experiments and discussion are depicted in Section 5. Section 6 provides a summary and a discussion of some extensions of the paper.

2. Related work

The locality-sensitive hashing algorithm, as the effective method to deal with the nearest neighbor search (NNS) problem, is widely used in many applications, like image retrieval,[10–13] classification,[14,15] copy detection,[16,17] association rule mining,[18] and three-dimensional (3D) object indexing.[19] The most significant feature of LSH is that the construction of hash functions in LSH is dependent on diverse similarity metrics. This issue has been extensively discussed and many LSH methods are designed for various similarity measures, such as the l_1 distance,[7] the l_p distance $(0 < p \leq 2)$,[10] the EMD distance,[20] the χ^2 distance,[21] the Cosine similarity,[22] the Mahalanobis distance,[23] the Jaccard coefficient and the Kernel functions,[11,24] and so on.

Although even LSH can obtain perfect theoretical results, there are also some drawbacks when using LSH in practice. The main restriction of the LSH scheme is its huge memory consumption. This limitation seems to be inevitable as LSH needs enough hash tables to maintain high recall and high precision. The hash tables increase with the growing data, and the index structure may not be stored into the main memory. Once the hash tables exhaust the main memory and overflow into the slow disk, we must execute the I/O operation resulting in an awfully slow execution speed. Facing this drawback, currently there are three strategies to reduce the storage space of LSH: the multi-probe strategy, the binary code's storage form, and the distribute LSH. Some variations of LSH based on the multi-probe strategy have been put forward. Entropy-based LSH[25] is the first one to utilize this strategy. Through hashing several random points which are chosen based on the entropy of the hash value of the current random point in the neighborhood of the query point, additional hash buckets will be probed and probed results are merged. The author also proved that at least one of the random points will be hashed into the bucket including the nearest neighbor. As a result, the candidate dataset will contain more points and the recall will be enhanced. In accordance with this scheme, fewer hash tables are needed, thus, less storage space is needed. Another method that takes advantage of the multi-probe strategy is the multi-probe LSH.[1] It is desired in the light of theoretical work on entropy-based LSH, which reduces the space requirement of the basic LSH method, and the query adaptive method.[26] Multi-probe LSH intelligently probes multiple buckets that are likely to contain query results in a hash table. It is proved that multi-probe LSH has a similar time complexity as the basic LSH method while using fewer hash tables by an order of magnitude. Gu *et al.*[27] proposed a novel probability model and a query-adaptive algorithm to generate the optimal multi-probe sequence for range queries. This method was proved to probe fewer points than multi-probe LSH for gaining the same recall. Unlike multi-probe LSH which has an unknown possibility to get the best performance, *posteriori* multi-probe LSH[28] brought forward a more reliable *posteriori* model taking some *a priori* knowledge about the queries and the searched objects into considerations. This *a priori* knowledge implements a better quality control of the search and helps to exactly select the most probable hash buckets. The above algorithms based on the multi-probe strategy are all proved to achieve the goal to reduce the memory consumption while keeping the comparable query precision and recall.

Apart from the multi-probe strategy, recently proposed methods are implemented on the basis of converting each point into a compact binary code, which are also proved to decrease the query time with much less storage. Representative methods following this strategy contain spectral hashing[29] and

semi-supervised hashing (SSH).[30] Spectral hashing performs nonlinear dimensionality reduction using principal component analysis to generate short binary codes. Semi-supervised hashing which is formulated as minimizing empirical error is proposed to learn efficient hash codes. This method can deal with semantic similarity among the data points. Wang[30] proved that semi-supervised hashing methods can solve the problem of overfitting when labeled data are small or noisy.

In addition to these two methods mentioned above, there are some other methods which depend on the same strategy stated above.[31,32] In the near future, many researchers have brought forward another novel technique to endure the LSH's space complexity. They combine the LSH scheme with the distributed system or parallel computing.[33] Koga *et al.*[34] introduced a simple and natural method to implement LSH in the distributed environment, which made every node keep the same number of hash tables. It successfully reduced remote accesses while it brought about multiple hash buckets being accessed in processing a query with a single remote access.

Another shortcoming which influences the LSH's best performance is the selection of parameters. Slaney *et al.*[35] described an algorithm for optimizing the parameters and use of LSH. The paper took advantage of two histograms to return the LSH parameters which made the LSH index satisfy the performance demand and have the minimum computational cost. LSH forest[36] improved the basic LSH method by eliminating the different data-dependent parameters for which LSH must be constantly hand-tuned. Meanwhile, this method optimized LSH's performance for skewed data distributions while maintaining the same storage space and query consumption. Authors also proved this method could be implemented in parallel systems and in peer-to-peer systems.

The limitations mentioned above have recently been thoroughly investigated. Gu *et al.*[37] detected a new flaw which was neglected before. They proposed that when the dataset is large-scale, the number of points to be filtered in the last step of LSH becomes large, which may be the bottleneck of improving the query speed when the amount of datasets becomes larger and larger. The paper dramatically improved the filtering speed using triangle inequality by overcoming the problem that arises from a "non-uniform" phenomenon.

The last drawback is that most of the existing LSH variations randomly generates hash functions which may cut down the performance. Choosing more appropriate hash functions to better partition the vector space plays an important role in the LSH's performance.[38] Cao *et al.*[14] brought forward two improved LSH algorithms based on a weakly supervised learning technique. The first one selected the most appropriate hash functions based on sample pairs labeled with "similar" or "dissimilar". The second one generated hash functions with positive sample pairs. The authors proved that both methods dramatically reduced both space complexity and time complexity.

According to the above discussion, we can conclude that the locality-sensitive hashing method is an effective index structure for large-scale and high-dimensional datasets. Nevertheless, the huge space requirements, the difficulty of selecting appropriate parameters, huge time consumption of candidate points, and the weakness of selecting apposite hash functions have a big influence on the performance of the locality-sensitive hashing method. In this paper, we bring about an improved p-stable LSH algorithm called RLSH for the purpose of reducing the space requirement and the time consumption.

3. Locality-sensitive hashing based on *p*-stable distributions

In this section, we will describe the locality-sensitive hashing based on p-stable distributions in detail. Firstly we briefly present the p-stable distributions. Then we detail the locality-sensitive hashing based on p-stable distributions for the preparation for our proposed algorithm.

3.1. *p*-stable distributions

In probability, a random variable is considered to have a stable distribution, if it has the property that a linear combination of two independent copies of the variables has the same distribution, up to location and scale parameters. The stable distribution is sometimes referred to as the Lévy alpha-stable distribution.[39] The normal distribution,[40] the Cauchy distribution,[41] the Lévy distribution,[42] and the heavy-tailed distribution[43] are all special cases of stable distributions.

Stable distribution[9]: a distribution $\mathcal{D}$ over $\Re$ is called p-stable, if there exists $p \geq 0$ such that for any n real numbers $v_1 \ldots v_n$ and independent and identically distributed variables $X_1 \ldots X_n$ with distribution $\mathcal{D}$, the random variable $\sum_i v_i X_i$ has the same distribution as the variable $(\sum_i |v_i|^p)^{1/p} X$, where X is a random variable with distribution $\mathcal{D}$.

Zolotarev[44] proved that the stable distributions exist for any $p \in (0,2]$. The following two kinds of stable distributions are often used in our computing vision.

1) A standard Cauchy distribution $\mathcal{D}_c$: it has the distribution of a random variable that is the ratio of two independent standard normal random variables. It is designed by the density function $f(x;0,1) = 1/(\pi(1+x^2))$, is 1-stable.

2) A Gaussian (normal) distribution $\mathcal{D}_G$: it is defined by the density function

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right],$$

is a 2-stable. The parameter μ in this formula is the mean or expectation of the distribution. The parameter σ is its standard deviation, and therefore, its variance is σ^2.

The Gaussian (normal) distribution $\mathcal{D}_G$ is used to generate the hash family in the p-stable locality-sensitive hashing.

3.2. *p*-stable locality-sensitive hashing

The principle of LSH is making use of multiple hash functions from a certain hash family, which satisfies the following locality-sensitive definition, to reduce the query range to the points stored in the same hash buckets as the query point. Finally, LSH only needs to compute the similarity between these points in the reduced query range and the query point. Hence, LSH successfully reduces the computational overhead.

Locality-sensitive hash family: a family $H = \{h : S \to U\}$ of functions over S is called (r_1, r_2, p_1, p_2)-sensitive if for any two objects p, q in S:

$$\text{if } \mathrm{d}(p,q) \le r_1 : \Pr_{h\in H}[h(p) = h(q)] \ge p_1, \tag{1}$$

$$\text{if } \mathrm{d}(p,q) \ge r_2 : \Pr_{h\in H}[h(p) = h(q)] \le p_2, \tag{2}$$

where $\mathrm{d}(p,q)$ is the distance between objects p and q. $\Pr_{h\in H}[h(p) = h(q)] = \mathrm{sim}(p,q) \in [0,1]$ is the similarity function defined on the current dataset. $h \in H$ is a hash function which projects a vector to an integer.

For the locality-sensitive hashing scheme, two restrictions should be satisfied: $r_2 > r_1$ and $p_2 < p_1$; $(1+\varepsilon) = r_2/r_1$ is the approximation factor. Obviously, the definition with the restrictions shows that nearby points within distance r_1 are more likely to collide ($p_1 > p_2$) than points with a distance greater than r_2.

For the purpose of decreasing the probability of false detection p_2 or enlarging the difference between the probabilities p_1 and p_2, multiple hash functions in a certain hash family are concatenated in the LSH scheme. For a given integer k, $g(p) = (h_1(p), h_2(p), \ldots, h_k(p))$ is generated from a hash family $G = \{g : S \to U^k\}$, where $h_i \in H$. For the hash family G, probabilities in expressions (1) and (2) must be revised as:

$$\text{if } \mathrm{d}(p,q) \le r_1 : \Pr_{g\in G}[g(p) = g(q)] \ge {p_1}^k, \tag{3}$$

$$\text{if } \mathrm{d}(p,q) \ge r_2 : \Pr_{g\in G}[g(p) = g(q)] \le {p_2}^k. \tag{4}$$

Intuitively, the collision probability of two close points ${p_1}^k$ also decreases, that is improper. To compensate for the decrease in this probability, several functions g are used. For a given parameter L, the LSH scheme independently and uniformly chooses $g_1, \ldots, g_L$ from G randomly. Each function g_i represents a new hash table, so we gain L hash tables.

Datar[10] proposed the locality-sensitive hashing scheme based on p-stable distributions which is suitable for the l_p ($p \in (0,2]$) norm. In this scheme, each hash function works on tuples of random projections of this form

$$h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{W} \right\rfloor,$$

where W means the quantization bucket width, a is a random vector whose entry is chosen independently from a p-stable distribution (say, $p = 2$, it is a Gaussian distribution), b is a real number selected uniformly in the range $[0, W]$ representing an offset.

After introducing the definition of p-stable LSH, we can conclude that there are three main parameters that influence the performance of p-stable LSH: the number of projections per hash value (k), the number of hash tables (L), and the width of the projection (W). Increasing W will increase the probability of collision for any two points. The parameter W represents a trade-off between a bigger table with a smaller final linear search or a more compact table with more points to consider in the final search. Varying k affects a trade-off between the time spent in computing hash values and the time spent in pruning false positives, i.e., computing distances between the query and the candidates. A bigger k will increase the number of hash computations, although the arithmetic operations are cheap compared with memory lookups. To fulfil that the probability of LSH failing to find the true nearest neighbor is less than ε, for a given value k, L is required to be at least $L = |\log \varepsilon| / (\log(1 - {p_1}^k))$.[45]

The procedure to find the nearest point to the query q from S involves the next two steps.

Step 1: Generation of hash tables

S1-1) Selecting tuples of the random vector (a,b): randomly select k vector a with the same dimensionality as the point in the dataset, where each coordinate is a Gaussian random variable $N(0,1)$, and an offset b generated from a uniform random distribution between 0 and W.

S1-2) One-line projection: select a point $p \in S$, compute its hash value with the hash function

$$h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{W} \right\rfloor.$$

S1-3) Multi-line projection: as we independently select k tuples of the random vector (a,b), we can obtain a list of k integers by executing k one-line projections. The LSH scheme holds the completion that points which project to the same k integers are members of the same hash bucket. Usually at this stage, a conventional hash function is used to convert the k-dimensional hash value to a one-dimensional integer. After this stage, part of the certain hash table is generated.

S1-4) Project the current point into L hash tables, and repeat S1-2) and S1-3) until the whole dataset has been scanned. Then L hash tables are completely constructed.

Step 2: Nearest neighbor search for the query q.

S2-1) Compute the L hash values for the query q taking advantage of the same tuples of the random vector, which is the same as Step 1.

S2-2) Let P_i be the set of points in S stored in the same bucket as in the hash table g_i. Then the points in

$P_1 \bigcup P_2 \bigcup \ldots \bigcup P_L$ become the candidates in the query range for the nearest point to q. Compute the distance between the query q and these candidates, and return the one which is closest to q.

3.3. Complexity analysis

After introducing the process of the p-stable LSH, we can conclude a simple computational model. $T_{\text{projection}}$ is denoted as the cost for a single projection of points in the dataset, T_{hash} is denoted as an upper bound of the cost for computing the hash value corresponding to the query point, while T_{distance} shows the cost of computing the distance between a candidate point and a query point. Thus, the time complexity of p-stable LSH is $nkLT_{\text{projection}} + kLT_{\text{hash}} + N_{\text{c}}T_{\text{distance}}$ and the space complexity is nL, where n is the number of objects in the current dataset, L is the number of hash tables, k is the number of projections per hash value and N_{c} is the amount of the candidate points after the first step of the p-stable LSH algorithm.

The p-stable LSH is one of the most popular species of LSH, so limitations of the basic LSH scheme equally exist in it. Our proposed algorithm-RLSH effectively reduces the space consumption and improves the query speed while maintaining the same precision and recall.

4. Improved locality-sensitive hashing

According to the discussion in Section 2, the traditional p-stable LSH algorithm has the following disadvantages. Firstly, it needs huge space requirements for storing sufficient hash tables aiming to keep high recall and high precision. Secondly, it requests users to specify three main parameters: the number of projections per hash value (k), the number of hash tables (L), and the width of the projection (W). However, there is no commonly applicable method for this algorithm of selecting appropriate parameters. A lot of researchers determine these parameters on the basis of the *a priori* knowledge about the dataset or recommended value by other literatures. Finally, the traditional p-stable LSH algorithm also has the potential bottleneck, that the candidate points become large with the increasing of data, which influences the performance of this algorithm.

In this paper, we improve the traditional p-stable LSH algorithm in terms of the huge space requirements and high time consumption. Currently, lots of researchers have put forward several improvement strategies in this aspect. These improvement strategies can be classified into two categories. The first one is selecting hash buckets with high success probabilities of containing the nearest neighbor instead of blindly searching enough hash buckets. Entropy-based LSH[25] and multi-probe LSH[1] are the most typical methods taking advantage of this strategy. Entropy-based LSH orders hash buckets depending on the nearest neighbor distance, that ensures all hash buckets with high probabilities are probed. Unlike entropy-based LSH, multi-probe LSH uses a carefully derived probing sequence to examine multiple buckets that are likely to contain the nearest neighbors of the query point. The other way to improve this problem of the traditional p-stable LSH algorithm is promoting the computing ability. Distributed system and parallel computing are the optimal choices to be integrated into the traditional p-stable scheme.[33,34]

So far, these improvements of the traditional p-stable algorithm have not taken accelerating the algorithm by reducing the number of projections for the query point into consideration. However, the probability of finding the nearest neighbors may be cut down in this way because there are fewer hash buckets to be searched. To solve this contradiction, a novel method of selecting the candidate points is brought forward. This method achieves the goal of containing more candidates which may include the nearest neighbors of the query point and simultaneously reducing the number of projections for the query point. More details about our proposed algorithm will be described in the following section.

4.1. Randomness-based locality-sensitive hashing scheme

This novel algorithm is similar to the traditional p-stable LSH except that it randomly selects a hash table to project the query point q while the traditional p-stable LSH algorithm projects the query point q to all the hash tables. The selection of candidates for these two methods is different too. RLSH works in the same way in the first step, i.e., the generation of hash tables as the p-stable LSH, so we describe the second step in detail.

Step 1: Generation of hash tables.

Step 2: Nearest neighbor search for the query point q.

1) Randomly select a hash table g_r from all the L hash tables, compute the hash value for the query point q taking advantage of the same tuples of the random vector in this selected hash table, which is the same as Step 1.

2) Let S_r be the set of points stored in the same bucket as q in the hash table g_r, scan points which are stored in the same hash bucket as points in S_r in the remaining hash tables $g_i\,(i \neq r)$.

3) Let P_i be the set of points in S stored in the same bucket as the points in S_r in the hash table $g_i\,(i \neq r)$. Then the points in $P_1 \bigcup P_2 \bigcup \cdots \bigcup P_L \bigcup S_r\,(i \neq r)$ become the candidates in the query range for the nearest point to q. Compute the distance between the query point q and these candidates, and return the one which is closest to q.

4.2. Complexity analysis

After comprehending the process of the RLSH algorithm, conclusions can be made that the time consumption is the main cost in the following three aspects: the projection of objects in

the dataset, the projection of the query point, and the computing of distances between the query point and the candidate points.

Considering the above notations, $T_{\text{projection}}$ is denoted as the cost for a single projection of points in the dataset, T_{hash} is denoted as an upper bound of the cost for computing the hash value corresponding to the query point, while T_{distance} shows the cost of computing the distance between a candidate point and a query point.

In the first step of the RLSH algorithm, each object in the dataset needs to carry out the one-line projection for k times to obtain k integers making use of k hash functions

$$h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{W} \right\rfloor$$

with different tuples of the random vector (a,b). Then a conventional hash function is used to convert the k-dimensional hash value to a one-dimensional integer. In this paper, we assume that a projection with these two kinds of hash functions has the same time consumption $T_{\text{projection}}$. The time complexity of this process is

$$T_{\text{one_hashtable}} = nkT_{\text{projection}}, \tag{5}$$

where k is the number of projections per hash value, and n is the amount of points in the dataset. To increase the points that collide with the query point, L hash tables are used. Generally speaking, the whole time complexity of the first step is

$$T_{\text{multiple_hashtable}} = nkLT_{\text{projection}}, \tag{6}$$

where L is the number of hash tables.

In the second step for searching the nearest neighbors of the query point, a certain hash table is chosen for mapping the query point. As mentioned above, T_{hash} is denoted as the cost for computing the hash value of the query point. So the time complexity is

$$T_{\text{query_hash}} = kT_{\text{hash}}, \tag{7}$$

where k is the number of projections per hash value.

After obtaining the candidate points, what we need to conduct is computing the real distance between the query point and the points in the candidate set. Hence, the time complexity of this procedure is

$$T_{\text{candidate}} = N_{\text{c}}T_{\text{distance}}, \tag{8}$$

where N_{c} is the number of candidate points. Thus, the time complexity of RLSH is

$$T_{\text{total}} = nkLT_{\text{projection}} + kT_{\text{hash}} + N_{\text{c}}T_{\text{distance}}. \tag{9}$$

In the procedure of the RLSH algorithm, the space consumption is used for storing hash tables and the dataset. So the space complexity is

$$S = nL, \tag{10}$$

where n is the size of the data set, and L is the number of hash tables.

We notice that the cost for computing the hash value of the query point is decreased to kT_{hash} and the space cost does not change at all. What we have to claim is that N_{c} of this algorithm is uncertain, which results in the difficulty of comparing with the time complexity of the traditional p-stable LSH algorithm. So this improvement theoretically is not dramatically obvious. However, in practice, the following experiments have shown that RLSH is more time-efficient than p-stable LSH.

In the aspect of recall and precision, the strategy of choosing the candidate points ensures that many points with high success probabilities are included in the candidate set of RLSH. Not only are the points in the same hash bucket as the query point in a randomly selected hash table contained in the candidate set, but also the points which collide with these above points in other hash tables are considered in the candidate set. The diversity of the candidate set is improved by this method. Overall, RLSH prevents the loss of the nearest neighbors of the query point as much as possible to guarantee the high recall and high precision. An additional superiority of this strategy is that RLSH may provide a higher probability of finding the nearest neighbor even with fewer hash tables for the purpose of reducing the space requirement because of the diversity of the candidate set.

5. Experiments and discussion

In this section, we describe the detailed information of our experiments, which consists of the evaluation datasets, evaluation benchmarks, evaluation metrics, and experimental results. The Java programming language is used. The evaluation is done on a PC with one Intel dual-processor 2.8 GHz CPU and 2 GB of DRAM.

5.1. Evaluation datasets

To evaluate the performance of our proposed algorithm, we utilize two different datasets. The synthetic dataset is designed by the random values satisfying the Gaussian distribution. The second dataset is an open dataset provided by the TEXMEX Research Team.[46] The TEXMEX Research Team provides several evaluation sets for us to evaluate the quality of the approximate nearest neighbors search algorithm on different kinds of data and varying database sizes. Both two datasets are summarized in Table 1.

Table 1. Test datasets summarizing.

Dataset	Objects	Dimensions	Total size/M
Synthetic dataset	65536	9	2.4
ANN_SIFT1M	10^6	128	316

5.2. Evaluation benchmarks

For each dataset, we randomly select 15 objects as the query points to become an evaluation benchmark. With regard to each query point, the ideal answer is defined as the query's K nearest neighbors (not including the query point itself default), based on the Euclidean distance. In our experiment, K is fixed as 10.

5.3. Evaluation metrics

According to the performance metrics mentioned in Section 2, the performance of the nearest neighbor problem can be evaluated in two aspects: the search speed and space requirement.

Search quality is usually measured by recall. Given a query point q, $I(q)$ is the result set of points obtained by the algorithm (i.e. the K nearest neighbors of q), while $A(q)$ is the set of actual answers, then

$$\text{recall} = \frac{|A(q) \bigcap I(q)|}{|I(q)|}. \tag{11}$$

Of course, the ideal recall is 1.0, that means all the K nearest neighbors are precisely returned.

When we refer to the search speed, the query time cost spent on a query is considered both by our algorithm and the traditional p-stable LSH under the same configuration, like the parameters K, L, and W mentioned above and the recall. In our experiment, the space requirement is equal to the memory to store the hash tables (i.e. the number of hash tables) and the memory required to store the data points.

5.4. Experimental results

The main result of our experiment is that our proposed algorithm RLSH is much more time-efficient and space-efficient than the traditional p-stable LSH while achieving the same recall. What we have to declare is that the query time consists of the time spent on the process of the locality-sensitive hashing method and the time spent on the procedure of computing the Euclidean distance between the query point and the candidate points. So when the dimension is larger, the time spent on the computing process is larger.

Figure 1 shows the distribution of the time consumption with a different number of projections per hash value (K). The other parameters are specified, $L = 45$, and $W = 4$. From this experiment, what we can conclude is that the parameter K plays a role as the tradeoff between the time spent in computing the hash values and the time spent in computing the distances between the query point and the candidate points in these two algorithms. A bigger K will result in a longer time consumption of hash computations. Different values of K will be set with different data sets. Currently, there is no better way to select the appropriate K. In the following experiments, the value of K is fixed as 10 in default, according to Ref. [9].

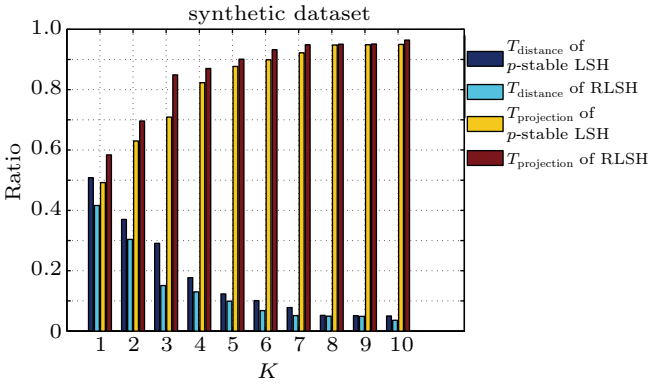


Fig. 1. (color online) The distribution of time consumption with different K.

Figure 2 reveals that RLSH is more time-efficient than the p-stable LSH with the increasing of the dataset when the size of the dataset reaches to 3.6×10^4. We randomly select ten datasets from the synthetic dataset. These datasets respectively contain 6×10^3, 1.2×10^4, 1.8×10^4, 2.4×10^4, 3.0×10^4, 3.6×10^4, 4.2×10^4, 4.8×10^4, 5.4×10^4, and 6.0×10^4 points. We can see that when the amount of points in the dataset is less than 3.6×10^4, our method spends a little more time than the p-stable LSH method. However, our method is better than the p-stable LSH method in terms of the query time when the number of points in the dataset is larger than 3.6×10^4. We conclude that our algorithm needs a shorter query time than the p-stable LSH under the circumstances of a larger size of databases. If the size of datasets is not very large, the superiority of our algorithm is not obvious. However, we find that the advantage of RLSH will become more and more apparent with the increase of the database size.

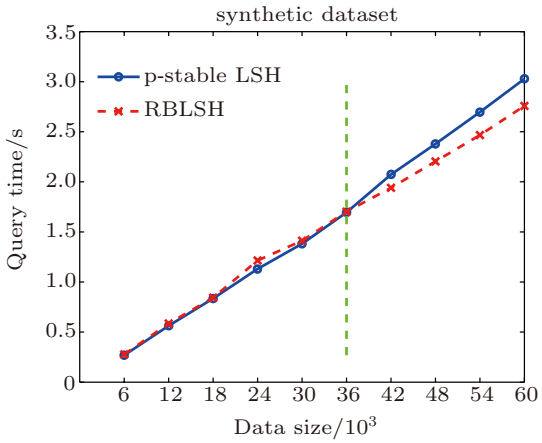


Fig. 2. (color online) Query time comparison with different data sizes.

Figure 3 shows that the speedup comparison between the p-stable LSH method and RLSH. The speedup is defined as the ratio of the query time of the p-stable LSH method and RLSH. According to Fig. 2, the speedup becomes larger than 1, that means RLSH is more time-effective than RLSH, when the size of the dataset reaches a certain level. As the sizes of the database increase, the speedup also improves.

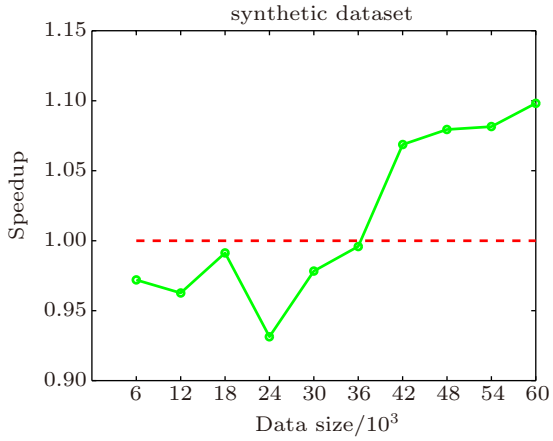


Fig. 3. (color online) Speedup comparison with different data sizes.

Table 2 represents the relations between the various search quality levels (i.e. recall) and the query time of both RLSH and the *p*-stable LSH. The query time is the average result of the *p*-stable LSH and RLSH using ten random queries with the synthetic dataset and the ANN_SIFT1M dataset. What we can obtain from this table is that RLSH is also superior to the *p*-stable LSH method with respect to different sizes of datasets.

Table 2. Query time comparison of different datasets.

	Synthetic Dataset		ANN_SIFT1M	
Recall	*p*-stable LSH	RLSH	*p*-stable LSH	RLSH
98%	3.544	3.350	67.863	61.876
95%	3.388	3.001	51.627	41.52
92%	2.431	2.267	47.522	29.596

At the end of our experiment, we also compare the space requirement of both these two algorithms. On the basis of the information shown in Fig. 4, these two points can be verified. On the one hand, the recall becomes higher when the number of hash tables increases. On the other hand, RLSH can achieve higher recall than the *p*-stable LSH method when they use the same number of hash tables, i.e., if we want to reach the same recall, RLSH is more space-efficient than the *p*-stable LSH method.

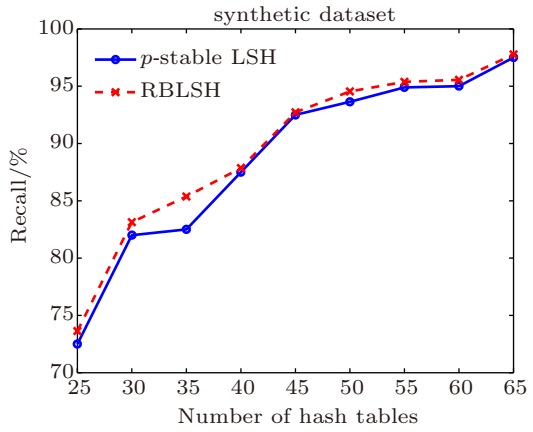


Fig. 4. (color online) Recall comparison with different number of hash tables.

6. Conclusion and future research

In our paper, we have completely described the theory of the popular locality-sensitive hashing, which is widely used in the approximate nearest neighbor problem. The LSH scheme is proved to be effective to deal with high-dimensional and large-scale datasets. However, it still has some limitations that are restrictions to obtaining the best performance, i.e., time efficiency and space efficiency. For the purpose of improving the performance of LSH, we propose a novel improvement algorithm, called RLSH, based on the traditional *p*-stable LSH. The experiments prove that our algorithm is more time-efficient than the latter while keeping the same recall. Moreover, RLSH reduces the number of hash tables also while maintaining the same recall. In the future, we will research these two issues: the selection of appropriate parameters and how to obtain more accurate candidate points. Although our algorithm has better performance, we did not consider how to select the best parameters for our algorithm. Another uncertainty about RLSH surrounds how to select candidates if there are no points in the same hash bucket as the query point when one randomly chooses a certain hash table. We plan to study these issues in the near future.

References

[1] Lü Q, Josephson W, Wang Z and Charikar M 2007 *Proceedings of the 33rd International Conference on Very Large Data Bases*, September 23–28, 2007, Vienna, Austria, p. 950
[2] Friedman J H, Bentley J L and Finkel R A 1977 *ACM Transactions on Mathematical Software* **3** 209
[3] Guttman A 1984 *Proceedings of ACM Conference on Management of Data* (SIGMOD), June 18–21, 1984, Boston, Massachusetts, p. 47
[4] Tsaparas P 1999 *Qualifying Depth Oral Report* 31902
[5] Jégou H, Douze M and Schmid C 2011 *IEEE Transaction Pattern Analysis and Machine Intelligence* **33** 117
[6] Lejsek H, Asmundsson F H, Jónsson B and Amsaleg L 2009 *IEEE Transaction Pattern Analysis and Machine Intelligence* **31** 869
[7] Indyk P and Motwani R 1998 *Proceedings of the 30th on Symposium on Theory of Computing* (STOC), May 23–26, 1998, Dallas, Texas, USA, p. 604
[8] Gionis A, Indyk P and Motwani R 1999 *Proceedings of the 25th on Very Large Data Bases* (VLDB), September 7–10, 1999, Edinburgh, Scotland, UK, p. 518
[9] Datar M, Immorlica N, Indyk P and Mirrokni V S 2004 *Proceedings of the 20th Annual Symposium on Computational Geometry*, June 8–11, 2004, Brooklyn, New York, USA, p. 253
[10] Liu F, Leung H Y, Cheng L M and Ji X Y 2012 *Chin. Phys. B* **21** 040204
[11] Cao Y, ZhangH and Guo J 2011 *Proceedings of the 18th International Conference on Image Processing*, September 11–14, 2011, Brussels, Belgium, p. 2461
[12] Liu F, Leung H Y, Cheng L M and Ji X Y 2012 *Chin. Phys. B* **21** 040204
[13] Hou J, Yan G F and Fan Z 2011 *Chin. Phys. B* **20** 048103
[14] Paula L B, Villaca R S and Magalhaes M F 2010 *Proceedings of the 4th International Conference on Semantic Computing* (ICSC), September 22–24, 2010, Pittsburgh, PA, USA, p. 408
[15] Ma J F, Hou K, Bao S L and Chen C 2011 *Chin. Phys. B* **20** 028701
[16] Zhang Z, Cao C, Zhang R and Zou J 2010 *Proceedings of the International Conference on Automation and Logistics* (ICAL), August 16–20, 2010, Hong Kong and Macau, China, p. 13
[17] Long M, Peng F and Chen G R 2008 *Chin. Phys. B* **17** 3588
[18] Chen C, Horng S and Huang C 2011 *Expert Systems with Applications* **38** 12388

[19] Matei B, Shan Y, Sawhney H S, Tan Y, Kumar R, Huber D and Hebert M 2006 *IEEE Transactions on Pattern Analysis and Machine Intelligence* **28** 1111

[20] Indyk P and Thaper N 2003 *Proceedings of the International Workshop Statistical and Computational Theories of Vision*, July 13–15, 2003, Vancouver, Canada, p. 33

[21] Gorisse D, Cord M and Precioso F 2012 *IEEE Transaction on Pattern Analysis and Machine Intelligence* **34** 402

[22] Charikar M S 2002 *Proceedings of the* 34*th Symposium on Theory of Computing* (STOC), May 19–21, 2002, Montréal, Québec, Canada, p. 380

[23] Jain P, Kulis B and Grauman K 2008 *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition* (CVPR), June 23–28, 2008, Anchorage, Alaska, USA, p. 1

[24] Kulis B and Grauman K 2012 *IEEE Transactions on Pattern Analysis and Machine Intelligence* **34** 1092

[25] Panigrahy R 2006 *Proceedings of the* 17*th Annual ACM-SIAM Symposium on Discrete Algorithm* (SODA), January 22–26, 2006, Miami, Florida, USA, p. 1186

[26] Jégou H, Amsaleg L, Schmid C and Gros P 2008 *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, March 31–April 4, 2008, Las Vegas, USA, p. 825

[27] Gu X, Zhang L, Zhang D, Zhang Y, Li J and Bao N 2012 *Proceedings of the* 13*th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, August 8–10, 2012, Kyoto, Japan, p. 3

[28] Joly A and Buisson O 2008 *Proceedings of the* 16*th ACM International Conference on Multimedia*, October 26–31, 2008, Vancouver, Canada, p. 209

[29] Weiss Y, Torralba A and Fergus R 2008 *Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems*, December 8–11, 2008, Vancouver, British Columbia, Canada, p. 134

[30] Wang J, Kumar S and Chang S 2010 *Proceedings of* 2010 *Conference on Computer Vision and Pattern Recognition (CVPR)*, June 13–18, 2010, San Francisco, USA, p. 3424

[31] Wang J, Kumar S and Chang S 2010 *Proceedings of the* 27*th International Conference on Machine Learning*, June 21–24, 2010, Haifa, Israel, p. 1127

[32] Liu W, Wang J, Kumar S and Chang S 2011 *Proceedings of the* 28*th International Conference on Machine Learning*, June 28–July 2, 2011, Bellevue, Washington, USA, p. 567

[33] Pan J and Manocha D 2011 *Proceedings of GIS*, November 1–4, 2011, Chicago, USA, p. 211

[34] Koga H, Oguri M and Watanabe T 2012 *Proceedings of the* 26*th International Conference on Advanced Information Networking and Applications*, March 26–29, 2012, Fukuoka, Japan, p. 175

[35] Slaney M, Lifshits Y and He J 2012 *Proc. IEEE* **100** 2604

[36] Bawa M, Condie T and Ganesan P 2005 *Proceedings of the* 14*th International World Wide Web Conference (WWW)*, May 10–14, 2005, Chiba, Japan, p. 651

[37] Gu X, Zhang Y, Zhang L and Zhang D 2013 *Signal Processing* **93** 2244

[38] Paulevé L, Jégoub H and Amsalegc L 2010 *Pattern Recognition Letters* **11** 1348

[39] Stable distribution, http://en.wikipedia.org/wiki/Stable_distribution, 2013

[40] Normal distribution, http://en.wikipedia.org/wiki/Normal_distribution, 2013

[41] Cauchy distribution, http://en.wikipedia.org/wiki/Cauchy_distribution, 2013

[42] Lévy distribution, http://en.wikipedia.org/wiki/L%C3%A9vy_distribution, 2013

[43] Heavy-tailed distribution, http://en.wikipedia.org/wiki/Heavy-tailed_distribution, 2013

[44] Zolotarev V M 1986 *Translation of Mathematical Monographs* **65** 270

[45] Slaney M and Casey M 2008 *IEEE Signal Processing Magazine* **25** 128

[46] Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/, 2013