Locality-Sensitive Hashing Scheme Based on Heap Sort of Hash Bucket

Bo Fang
School of computer science and technology
Harbin Institute of Technology , shenzhen
Shenzhen, China
Email: fangbo@stu.hit.edu.cn

Zhongyun Hua
School of computer science and technology
Harbin Institute of Technology , shenzhen
Shenzhen, China
Email: huazhongyun@hit.edu.cn

Hejiao Huang*
School of computer science and technology
Harbin Institute of Technology , shenzhen
Shenzhen, China
Email: hjhuang@aliyun.com

***Abstract*—Nearest neighbor search (NNS) is one of the current popular research directions, which widely used in machine learning, pattern recognition, image detection and so on. In the low dimension data, based on tree search method can get good results. But when the data dimension goes up, that will produce a curse of dimensional. The proposed Locality-Sensitive Hashing algorithm (LSH) greatly improves the efficiency of nearest neighbor query for high dimensional data. But the algorithm relies on the building a large number of hash table, which makes the space complexity very high. C2LSH based on dynamic collision improves the disadvantage of LSH, but its disadvantage is that it needs to detect the collision times of a large number of data points which Increased query time. Therefore, Based on LSH algorithm, later researchers put forward many improved algorithms, but still not ideal.**

In this paper, we put forward Locality-Sensitive Hashing Scheme Based on Heap Sort of Hash Bucket (HSLSH) algorithm aiming at the shortcomings of LSH and C2LSH. Its main idea is to take advantage of the efficiency of heapsort in massive data sorting to improve the efficiency of nearest neighbor query. It only needs to rely on a small number of hash functions can not only overcome the shortcoming of LSH need to build a large number of hash table, and avoids defects of C2LSH. Experiments show that our algorithm is more than 20% better than C2LSH in query accuracy and 40% percent lower in query time.

***Index Terms*—Locality Sensitive Hashing, Nearest Neighbor Search, Heapsort.**

I. Introduction

The nearest neighbor search (NNS) in Euclidean space has been widely used in machine learning, information retrieval, pattern recognition and other fields. The definition of an NNS problem is that given a query point q and dataset D return nearest neighbor data points which are most similar to the query point q from the dataset. In order to solve the nearest neighbor search problem, many methods have been proposed, for example, R-tree[1], K-D tree [2] and SR-tree [3], they can achieve good query efficiency in medium and low dimensional data. With the increase of data dimension, the efficiency of these methods in the nearest neighbor query is greatly reduced, even worse than linear search [4], [5]. To solve this problem, in 1999, P. Indyk and R. Motwani proposed Locality-Sensitive Hashing (LSH) based on hamming distance [6]. The basic idea of LSH is that if two adjacent data points in the original data space are projected by the same projection or projection, it is highly likely that they will still be adjacent in the new data space, while it is highly unlikely that non-adjacent data points will be mapped to the same bucket. LSH compares the hamming distance of the hash value obtained, but generally the distance is measured by Euclidean distance. It is troublesome to map the Euclidean distance to the hamming space and then compare the hamming distance. Therefore, the researchers proposed Locality-Sensitive Hashing based on p-stable distribution (E2LSH) [7], which can directly deal with Euclidean distance.

For general LSH algorithms, we need to build L hash table, each hash table contains k hash functions, and then map all the data points of dataset to the hash tables, the data points which are stored in the same hash bucket as query point q will be the candidate neighbor points. Then, compute the distance of query point and the candidate point. Finally, the nearest data point to the query data point is returned. The proposed p-stable LSH can directly work on the Euclidean space without any embedding. This algorithm can find the nearest neighbor in time. Compared with previous methods, LSH performs very well in high-dimensional space. Although LSH can achieve good results in theory, it has some shortcomings in practical application. The main limitation of the LSH scheme is its large memory consumption. Because LSH requires enough hash tables to maintain query efficiency [8].

To overcome the shortcomings of LSH, researchers have proposed many improved methods. Entropy based LSH [9] is the earliest improved method. According to the entropy of the hash value of the current random point in the neighborhood of the query point, hash multiple random points, hash multiple hash buckets, and merge the hash results. As a result, candidate dataset will contain more points, and recalls will be enhanced. Under this scheme, fewer hash tables are required and less storage space is required. But the downside is that it's easy to get duplicate hash buckets. According to the entropy - based LSH theory, the space requirement of basic LSH method and query adaptive method is reduced.[10] Multi-probe LSH smart probe may contain multiple buckets of query results in the hash

978-1-7281-1846-8/19/$31.00 ©2019 IEEE

table. It proved that the Multi-probe LSH method has similar time complexity to the basic LSH method, and the number of hash tables used is reduced by order of magnitude. RLSH [11] solves the difficulty of generating neighbor data based on entropy LSH through random projection. Unlike Multi-probe LSH, whose possibility of obtaining the best performance is unknown, the posterior Multi-probe LSH [12] considers the prior knowledge of query and search object and proposes a more reliable posterior model. This prior knowledge enables better quality control of the search and helps select the most likely hash bucket accurately. The above method mainly reduces the number of hash tables by expanding the search scope, however, the effect is not very significant.

In order to significantly reduce the number of hash tables, C2LSH proposed a collision detection method. The idea of C2LSH [13] is to create L hash tables, each hash table has only one hash function, which can greatly reduce the number of hash tables, and then calculate the number of collisions to compare the hash values between data points. However, the disadvantage of this method is that it needs a large number of collision detection. LazyLSH [14] uses a single base index to support the computations in multiple Lp spaces, significantly reducing the maintenance overhead. Extensive experiments show that LazyLSH provides more accurate results for approximate KNN search under fractional distance metrics.

In this paper, we draw on the LazyLSH multi-paradigm and the idea of collision detection in C2LSH. We propose the HSLSH algorithm. In HSLSH, we establish a hash table, it contains the L hash function, we made up of L hash function value vector as a hash bucket ID, and then using heapsort to sort the hash bucket ID. The order is based first on the number of collisions and then on the distance between Manhattan. Heapsort greatly reduces the number of collision detection. Experiments show that HSLSH is much better than C2LSH in query accuracy and query time.

The rest of the paper is organized as follows. Section II introduces the nearest neighbor search. Section III introduces Locality-Sensitive Hashing. Section IV introduces C2LSH algorithm. Section V describes HSLSH in detail. Section VI is the experiments section. Section VII is the conclusion of this paper.

II. K-NEAREST NEIGHBOR SEARCH

In similarity search, data is described as high dimension vectors. When given query points and dataset as well as similarity measurement functions, the nearest neighbor search is to search data points most similar to query points from the dataset. Its formal description is as follows:

Give a query data q, A is denoted a set of returned data points which size is, denote the similarity measurement functions.

$$KNN(q) = A, \{A \subseteq X, \mid A \mid = K, \forall x \in A, \\ y \in X - A, D(q,x) \leq D(q,y)\} \quad (1)$$

K-nearest neighbor search can control the number of expected return points, so it is commonly used in multimedia information retrieval and other fields. In this paper, we use KNN to analyze our algorithm.

III. LOCALITY SENSITIVE HASHING(LSH)

The basic idea of LSH is as follows: First hashing the point which in the datasets, so that the probability of a collision between two points close to each other is much higher than two points far from each other. In the query, the query point is hashed into the bucket according to the same hash function, and then all points in the bucket are taken out as candidate approximate nearest neighbor points. Finally, the distance between the query point and each candidate approximate nearest neighbor point is calculated to determine whether it meets the query conditions.

1)if $D(p,q) \leq r_1$,then $P_{r_H}[h(p) = h(q)] \geq p_1$

2)if $D(p,q) \geq r_2$,then $P_{r_H}[h(p) = h(q)] \leq p_2$

In [6], the concept of function family is proposed, let S denote a set of d dimensions data points, $D : S \times S \to R$ is the similarity measurement functions, p and q are two arbitrary data points in S. Function family $H = \{h : S \to U\}$is - sensitive. If and only if :(in this $r_1 < r_2, p_1 < p_2$).

A. p-stable distributions

In [6], which has proposed p-stable LSH, we first introduce the p-stable distribution .

Denote: For the distribution D over a real number set R, if exist $P \geq 0$,for any n real numbers $v_1, v_2, ..., v_n$ and n variables that satisfy D distributions $X_1, X_2, ..., X_n$. Then the random variable $\sum_i v_i X_i$ and $(\sum_i \mid v_i \mid^p)^{1/p}$ have the same distribution, where X is a D distributed random variable then D is called a p- stable distribution.

For any p in $(0, 2]$,there is a stable distribution:

When $p = 1$ is Cauchy distribution, the probability density function is $c(x) = 1/[\pi(1 + x^2)]$.

When $p = 2$ is Gaussian distribution, the probability density function is $g(x) = 1/(2\pi)^{1/2} \times e^{-x^2/2}$.

The hash function of p-stable LSH is in formula (2):

$$h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{w} \rfloor (h_{a,b}(v) : R^d \to N) \quad (2)$$

Where a is a d-dimensional vector, each of which is a random variable independently selected from a p-stable. b is a random number within the range of $[0, w]$.

For two data points v_1, v_2,let $s = \mid\mid v_1, v_2 \mid\mid$. The probability that v_1 and v_2 collide under a uniformly randomly chosen hash function $h_{a,b}$,denoted as $p(s)$,can be computed in formula (3) :

$$p(s) = P_{r_{a,b}}[h_{a,b}(v_2)] = \int_0^w \frac{1}{s} f_2(\frac{t}{s})(1 - \frac{t}{w})dz \quad (3)$$

Where $f_2(x) = \frac{2}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$, the collision probability $p(s)$ decreases monotonically with s for a fixed w ,so the family of hash functions $h_{a,b}$ is sensitive with p_1, p_2.

Denote G ***function***: $g_i(v) = (h_1(v), .., h_k(v)), 1 < i < l$, that corresponds to L hash tables, and each function $g_i(\cdot)$ is generated independently. Each function $g_i(\cdot)$ consists of k hash functions which randomly selected independently from the hash function family $h_{a,b}$. The value of the function $g_i(\cdot)$ corresponds to a specific hash bucket.

The process of RNN or KNN search using LSH is mainly divided into two steps: index establishment and query. When setting up the index, for every point $v \in p$, calculate its L function values and store them in the corresponding hash bucket in tables. When querying, calculate L function values of query point q, then find L hash buckets where q is, calculate the distance between the points in these hash buckets and q, find the points within the specified distance (RNN), or k nearest points (KNN).

Complexity analysis: For each query structure, we create L hash tables, and each hash table has k hash functions. The calculation of time complexity is mainly divided into three parts. The first part is the time $T_{projection}$ to project the query point to the hash table, $T_{projection} = nklT_{hash}$ (T_{hash} denotes the time to project one hash function), the second part is the time T_{query} to return the candidate neighbor set,$T_{query} = klT_{search}$ (T_{search} denotes the time to traverse the data points in a hash function bucket) and the third part is the time $T_{calculate}$ to return the K nearest neighbor points by calculating the distance from the points in the candidate set $T_{calculate} = N_cT_{distance}$ (N_c denotes the number of candidate set, $T_{distance}$ is the time of calculating the distance of two data points).

So, $T = T_{projection} + T_{query} + T_{calculate} = nklT_{hash} + klT_{search} + N_cT_{distance}$. The space complexity is mainly determined by the index structure, S denotes space complexity, S is also divided into two parts, the first is the consumption of the original data set $O(dn)$,and the second for storing hash function is $O(dkl)$.

IV. LSH BASED ON DYNAMIC COLLOSION COUNTING (C2LSH)

Traditional LSH requires a large number of indexes to be built, it consumes space and increases query time, so in [8],C2LSH exploits only a single base ß of m LSH function$h_1, ..., h_m$, m is the size of base ß. each Hash table T_i is built by $h_i(\cdot)$, so,each data points o in the dataset D is hashed to an integer by $h_i(\cdot)$,which is taken an the bucket ID (or simply bid) of o. Then, all the data points are sorted in increasing order of their $bids$ along the real line. In other words, each hash table T_i is indeed a sorted list of buckets, and each bucket contains a set of object IDs representing the objects that fall in the bucket. If two objects are hashed by $h_i(\cdot)$ into the same bucket, we say they collide with each other under. To search NN for a query point q, C2LSH only considers distance computation for data objects that collide with q under a large enough number of functions in ß.

Complexity analysis: the query time of C2LSH consists of three parts, in this paper $m = O(\log n)$ (n is size of dataset). The first part is that time to locate the hash bucket corresponding to query object q is $md = O(d \log n)$,the second part ,collision detection time is $O(n \log n)$. Then,the calculation time of the candidate's real distance is $O(d)$. Query time complexity of C2LSH algorithm is $O(d \log n + n \log n)$. The total space complexity is $O(dn)$. Index space complexity is $O(n \log n)$.

V. LSH BASED ON HEAP SORT OF HASH BUCKET(HSLSH)

A. Introduction

The disadvantage of basic LSH is that a large number of hash tables need to be built to meet the query accuracy. Although C2LSH improves this disadvantage, it increases the query time complexity due to the need for a large number of collision detection. In order to solve these two contradictions, we propose HSLSH.It can avoid the disadvantages of LSH and use heapsort method to solve the disadvantages of C2LSH query time increasing.

In this paper, on the basis of the previous research, we put forward an algorithm with higher query efficiency. In C2LSH, due to the need of collision detection of a large number of data objects, the query efficiency is greatly reduced. Moreover, the size of the collision threshold is difficult to control. In this paper, instead of using a hash value vector as the bucket ID, we're going to use the vector $h_1(\cdot), ..., h_m(\cdot)$ from the value of the hash function of the entire hash table as the bucket ID. Then, based on the given vector comparison rules, use the heap method to build the heap, each time the heap is built, an optimal bucket is returned. The biggest advantage of heap sort is that it is much more efficient than other sorting methods to return the optimal data in the sorting of massive data, In the aspect of Multi-probe, our algorithm also shows good efficiency. Our method only needs to perform a pop-heap operation to expand the number of buckets detected, and the quality of the data in the bucket is better than Multi-probe LSH according to the given sorting rules. The following is a detailed description of HSLSH.

B. Algorithm Description

This algorithm randomly select hash functions from the family of hash functions as one hash table, each data object of data set D will get a hash value vector. Then, use as the bucket id of the hash bucket that holds the data object o. The detailed algorithm is divided into following steps:

Step 1: Generates a hash table G containing L hash functions.

Step 2: Map all the data objects to the corresponding bucket of the hash table.

Step 3: Return K Nearest neighbor search for the query point q. This step can divide into two part:

1) Map the query point to the hash table G, get the bucket id of the bucket, call it $tarVec$.

2) Heapsort bucket ids according to the given rule, as follows:

Step 1: Calculate the equivalent number of bucket vector elements C_t.

Step 2: If the C_t are equal and their Manhattan distances are compared.

3) Sets the ratio (r_i) that returns the number of buckets, Then use heapsort to return the optimal N_t buckets(N_t denoted the number of returned bucket), and use all data points which is stored in the returned bucket as candidate neighbor set S_c.

4) Calculate the Euclidean distance between data object and query object in the candidate neighbor set S_c and return the optimal K data objects.

Algorithm 1 and Algorithm 2 are the pseudo-codes of the hash table construction and the query process respectively.

Algorithm 1: HSLSH algorithm to generate hash tables

Input: Dimension d, The width of the hash bucket w; The size of hash table L; The Dataset D.
Output: The Hash tabel G, BID_s

$G = \emptyset$, $hft = \emptyset$, $BID_s = \emptyset$, BID_s denote the set of id of all data object;
$hft(d, l, w)$ *//The hft represents an array of hash functions objects*;
for $j = 1; j \leq L; ++i$ **do**
    $hf(d, w)$ *//generate a hash function object*;
    hft.push(hf);
//Generate its bucket id sequence for each data object;
for $j = 1; j \leq D.length; ++i$ **do**
    for $i = 1; i \leq L; ++i$ **do**
        $D[j].bid$.push($hft[i](D[j].data)$);
    $G[D[j].bid]$.insert($D[j].id$);
    BID_s.push($D[j].bid$);
return G, BID_s;

C. C.Algorithm Complexity Analysis

Time complexity: Similarly, the time complexity is divided into three parts: the first part is the indexing time ($T_{projection}$), the second part is the time(T_{search}) to return the candidate neighbor set, and the third part is the computing time $T_{compute}$. So $T_{total} = T_{projection} + T_{search} + T_{compute} = O(ldn) + O(N_t)l \log n + O(C \log C d)$($n$ is the size of D, d is dimension of data, C is the number of data points in the candidate dataset).

Space complexity: Space complexity is divided into two parts. The first is the space used to store the entire dataset is $O(nd)$ and second the space is needed to store the entire index structure is $O(nl)+O(n)$,so, $S_{total} = O(nd)+O(nl)+O(n)$.

VI. EXPERIMENT AND ANALYSIS

In this chapter, we will evaluate the performance of HSLSH algorithm through real data sets. We will compare HSLSH algorithm with the latest C2LSH algorithm. All algorithms use C++11 language, and all experiments are completed on machines with CPU Inter(R) Core(TM) i7-6700 CPU @ 3.4GHz,3,41GHz memory size of 8G.

Algorithm 2: HSLSH algorithm for query

Input: Query object q; Hash table G; The number of optimal buckets N_t; The id set ID_s
Output: KNN,The optimal K data object;

candidate set $S_c = \emptyset$; $B = \emptyset$; $KNN = \emptyset$;
for $i = 1; i \leq L; ++i$ **do**
    $q.id$.push($hft[i](D[j].data)$);
    for $j = 1; i \leq N_t; ++i$ **do**
        $optimal_{bid}$ =heapsort($BID_s, q.bid$);
        BID_s.delete($optimal_{bid}$);
        B.push($optimal_{bid}$);
for $i = 1; i \leq B.length; ++i$ **do**
    for $j = 1; j \leq G.length; ++j$ **do**
        S_c.push($G[B[i][j]$);
Sort(S_c,Euclidean distance(q, o in S_c));
for $i = 1; j \leq K; ++i$ **do**
    KNN.push($S_c[i]$);
return KNN;

A. Dataset

We will use two real data sets, *Mnist* and Fashion, that are commonly used to evaluate the performance of existing LSH algorithms.

Mnist. The *Mnist* dataset contains 60000 784-dimensional data objects, each of which is a handwritten digital image of size 28×28. The *Mnist* dataset also contains a test set of 10000. We randomly selected 50 data objects from the test set as the query set.

Fashion. *Fashion* is also contains 60000 784- dimensional data object, but compared with the previous *Mnist* datasets, a data object in each dimension relatively full, each data object is a size of 28×28 clothing pictures, at the same time, the data set includes a size of 10000 set of tests, we randomly selected 50 data object as a query set.

B. Evaluation of measurement

In our experiment, we evaluated the performance of the algorithm from the following two aspects

Query Efficiency: Query efficiency is mainly reflected in query time. We observe and compare the time of various algorithms by controlling the same variables.

Query Accuracy. In our experiment, we use approximate proportion to measure the quality of the algorithm query results. In particular, for each query object q, we compare the K-nearest neighbor returned by them with the real label of query object q, and use γ_i to represent the accuracy of each object q_i ,as shown in formula 4. R represents the average accuracy of the entire test set, as shown in formula 5.

$$\gamma_i = \frac{C_i}{K} \tag{4}$$

$$R = \frac{\sum_{i=1}^{m} \gamma_i}{m} \tag{5}$$

(C_i denoted the same number of neighbor objects as the real label, m is the number of test set).

C. Experimental parameter configuration

In this section, we discuss the impact of different parameter Settings on the performance of HSLSH algorithm. A large number of experimental results show that both HSLSH and C2LSH can achieve the highest query efficiency when hash functions $L = 10$ and bucket width $w = 200$. Then, we fixed the number of hash functions $L = 10$, and set the bucket $w = 200$ in the experiment. We run the program with the same number of threads as the test set data. In our experiment, the nearest neighbor K is a variable parameter, and the value of K is $10, 20..., 100$.

D. Experimental results

We first set $K = 10$ to find out the optimal parameters of C2LSH and HSLSH(the balance between query time and query precision is considered comprehensively), and then set different K values according to the optimal parameters r_i, and then compare the query efficiency of the two methods.

Figures $1-4$ shows the relationship between parameter adjustment(r_i and b) on the *Mnist* dataset and query time and query precision.

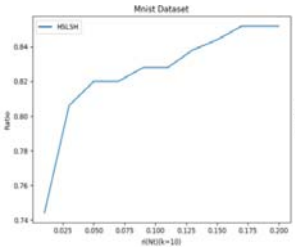

Fig. 1:

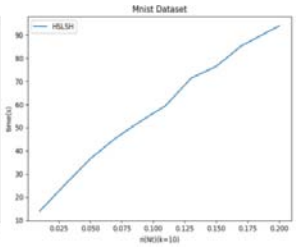

Fig. 2:

v

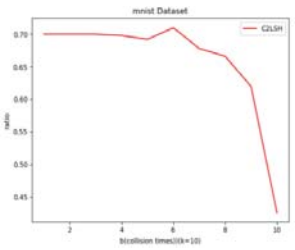

Fig. 3:

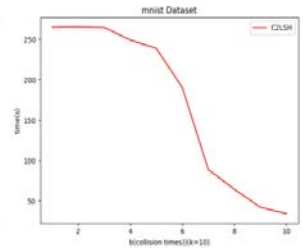

Fig. 4:

It can be observed from the figure above that, on *Mnist*, the optimal parameter $r_i = 0.05$ for HSLSH algorithm and $b = 8$ for C2LSH algorithm.

Figures $5-8$ shows the relationship between parameter adjustment(r_i and b) on the *Fashion* dataset and query time and query precision.

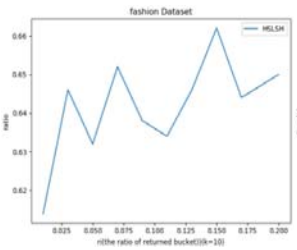

Fig. 5:

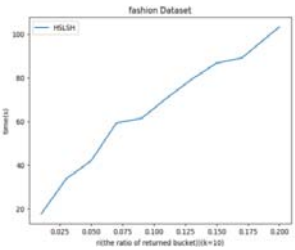

Fig. 6:

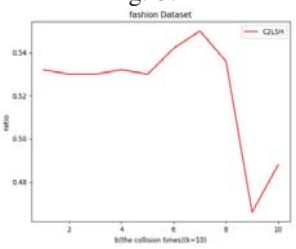

Fig. 7:

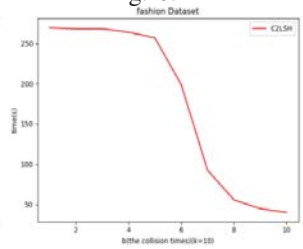

Fig. 8:

It can be observed from the figure above that, on *Fashion*, the optimal parameter $r_i = 0.03$ for HSLSH algorithm and $b = 8$ for C2LSH algorithm.

Next, we use the optimal parameters and observe the query efficiency of the two methods according to the different values.

Figure 9,10 shows the comparison of query efficiency between HSLSH and C2LSH on *Mnist*.

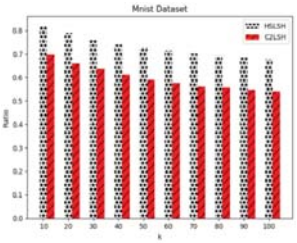

Fig. 9:

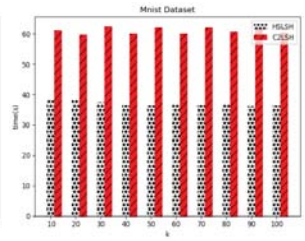

Fig. 10:

Figure 11,12 shows the comparison of query efficiency between HSLSH and C2LSH on *Fashion*.

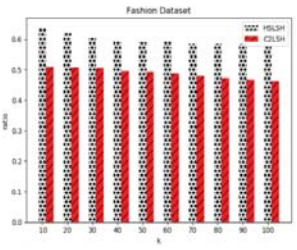

Fig. 11:

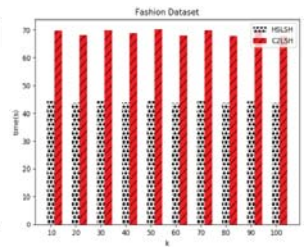

Fig. 12:

On the *Mnist* dataset, we can clearly see that the query accuracy of H2LSH is more than 20% higher than that of C2LSH under different K values, and the query time is more than 40% lower than that of C2LSH. Similarly,on the Fashion dataset the query accuracy of H2LSH is 20% higher than that of C2LSH, and the query time is about 40% lower than that of

C2LSH. To sum up, we can conclude that the performance of HSLSH algorithm in K-nearest neighbor query is much better than that of C2LSH algorithm.

VII. CONCLUSION

In this paper, we describe the HSLSH algorithm in full detail. HSLSH algorithm makes up for the deficiency of traditional LSH and C2LSH, and it greatly improves the query time and precision of K-nearest neighbor query by using heapsort in massive data query.When the data set is sufficiently large, the performance of HSLSH algorithm will be more efficient than that of C2LSH algorithm.

ACKNOWLEDGMENT

This work is financially supported by National Key $R\&D$ Program of China under Grant No. 2017YFB0803002 and No. 2016YFB0800804, National Natural Science Foundation of China under Grant No. 61672195 and No. 61732022.

REFERENCES

[1] Guttman. *R-tree: A dynamic index structure for spatial searching. In SIGMOD*, Pages 47-57,1984. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68C73.

[2] J. L.Bentley. *K-D trees for semi-dynamic points sets In Symposium on Computational Geometry*,1990.K. Elissa, Title of paper if known, unpublished.

[3] Katayama N and Satoh S. *The SR-tree: An index structure for high-dimensional nearest neighbor queries. SIGMOD*,1997.

[4] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. 2006.

[5] Y. Tao, K. Yi, C. Sheng, and P .Kalnis. *Efficient and accurate nearest neighbor and closest pair search in high dimensional space. ACM TODS*, 35,2010

[6] P. Indyk and R. Motwani. Approximate Nearest Neighbor: *Towards Removing the Curse of Dimensionality. In Proceeding of the 30th Symposium on Theory of Computing*.1998.pp.604-613.M. Young, The Technical Writers Handbook. Mill Valley, CA: University Science, 1989.

[7] M. Datar,N. Immorlica, P.Indyk, and V. Mirrokni. *Locality-sensitive hashing scheme based on p-stable distributions. Proceedings of the ACM Symposium on Computational Geometry*,2004.

[8] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. *Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. In PVLDB*, volume 6, pages 1930C1941, 2013

[9] Panigrahy R 2006 *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA)*, January 22C26, 2006, Miami, Florida, USA, p. 1186.

[10] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li.*Multi-probe LSH: efficient indexing for high-dimensional similarity search. In VLDB*, pages 950C961, 2007

[11] Lu Ying-Hua, Ma Ting-Huai, Zhong Shui-Ming, Cao Jie, Wang Xin and Abdullah Al-Dhelaane *Improved locality-sensitive hashing method for the approximate nearest neighbor problem*.2014.

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

[13] J. Gan, J. Feng, Q. Fang, and W. Ng, *Locality-sensitive hashing scheme based on dynamic collision counting, in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD* 2012, Scottsdale, AZ, USA, May 20-24, 2012, 2012, pp. 541C552.

[14] Y. Zheng, Q. Guo, A. K. Tung, and S.Wu. *Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In SIGMOD*, 2016.