Frequency Based Locality Sensitive Hashing

Kang Ling, Gangshan Wu
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
lingkang1988@gmail.com, gswu@nju.edu.cn

Abstract—**Nearest Neighbor (NN) search is of major importance to many applications, such as information retrieval, data mining and so on. However, finding the NN in high dimensional space has been proved to be time-consuming. In recent years, Locality Sensitive Hashing (LSH) has been proposed to solve Approximate Nearest Neighbor (ANN) problem. The main drawback of LSH is that it requires quite a lot of memory to achieve good performance, which makes it not that suit for today's application of massive data. We analyze generic LSH scheme as well as the properties of LSH hash functions based on *p*-stable distributions and propose a new LSH scheme called Frequency Based Locality Sensitive Hashing (FBLSH). FBLSH just uses one function based on *p*-stable distributions as hash function of a hash table, and it sets a frequency threshold *m*, only those points which collide with query point more than *m* times can be candidate ANNs. FBLSH is easy to implement and through experiments, we show that FBLSH can reduce the extra space cost by several orders of magnitude with less (or similar) time cost while achieving better search quality compared with LSH based on *p*-stable distributions.**

Keywords-Information Retrieval; Similarity Search; LSH

I. Introduction

Nearest Neighbor (NN) search is widely used in many applications, such as information retrieval, data mining, signal processing and so on. Some indexing methods using branch and bound techniques have good performance in low-dimensional space, KD-tree [1] for example. However, when it comes to high dimensional space, current techniques for NN search are proved to be no better than linear search, which is known as "curse of dimensionality" [2].

Some researchers proposed to improve the time efficiency by applying Approximate Nearest Neighbor (ANN) [3] search in recent years, because ANN can act as good as the exact one in many applications. Locality Sensitive Hashing (LSH) [3, 4] is one of the most popular ANN algorithms, which can achieve good performance in high-dimensional space. However, there is a trade-off between search efficiency and memory usage. To achieve high search efficiency, LSH requires quite a lot of memory due to a large number of hash tables used. But the data we deal with today is massive in many cases, the search quality and search efficiency on large dataset will be affected when the main memory capacity can not satisfy the space requirement.

A lot of improvements of LSH have been done recently. [5] proposed a near-optimal LSH that uses a Leech lattice for hashing in order to get better results than just using random projections. Query-adaptive LSH was proposed in [6], this method uses E8 lattices as hash functions and selects the most appropriate hash functions by a relevance criterion, resulting in less query time at the cost of more memory. In [7], entropy-based LSH selects several points in the neighborhood of query point and merges the query results of them. Obviously, it is a way of trading time for space. To reduce the time cost of entropy-based LSH, [8] proposed multi-probe LSH which searches several buckets that are likely to contain the query results for a query point. A posteriori multi-probe LSH has been proposed in [9], which defines a more reliable posteriori probabilistic model taking account some prior about the queries and the searched points.

These improvements are all based on generic LSH scheme. However, generic LSH scheme has some limitations as it trades space for time and does not make full use of hash functions. In this paper, we give the analysis of generic LSH scheme and present a new LSH algorithm scheme called Frequency Based Locality Sensitive Hashing (FBLSH) based on *p*-stable LSH. FBLSH does not use hash function which is generated by concatenating several functions chosen randomly from LSH family but just uses one function based on *p*-stable distributions. Correspondingly, it does not choose those points that collide with query point at least once as candidate ANNs but sets a frequency threshold *m*, only those points which collide with query point more than *m* times can be candidates. FBLSH can improve space efficiency and search quality with less (or similar) query time. Through experiments, we can prove the improvement over LSH based on *p*-stable distributions.

II. Background

A. Approximate Nearest Neighbor Search

What LSH deals with is Approximate Nearest Neighbor search problem, more specifically, (r, δ)-NN problem. Let X be the data domain and D: $X \times X \rightarrow R$ be the similarity function, we can define it as follows: given a query point q ($q \in X$), each point p ($p \in X$) satisfying $D(p, q) \leq r$ has to be returned with a probability at least δ.

B. Generic Locality Sensitive Hashing Scheme

The basic idea of LSH is to hash the points from the database so as to ensure that the probability of collision is

This work is supported by the National Science Foundation of China under Grant Nos. 61021062.

much higher for those points that are close to each other than for those that are far apart.

To implement LSH, the definition of LSH family was introduced in [3]. Let S be the domain of d-dimensional points and D: $S \times S \rightarrow R$ be the similarity function. A family $H = \{h : S \rightarrow U\}$ is called (r_1, r_2, p_1, p_2)-sensitive for D if for any $p, q \in S$:

- if $D(p, q) \leq r_1$ then $Pr_H[h(p) = h(q)] \geq p_1$,
- if $D(p, q) \geq r_2$ then $Pr_H[h(p) = h(q)] \leq p_2$.

To make LSH family be useful, it has to satisfy inequalities $r_1 < r_2$ and $p_1 > p_2$.

Generic LSH scheme works as follows:

1) Index construction: For an integer k, define a function family $G = \{g : S \rightarrow U^k\}$ such that $g(v) = (h_1(v), \ldots , h_k(v))$. For an integer L, choose L functions $g_1, \ldots , g_L$ from G, independently and uniformly at random. For any point v in the input dataset, store it in the bucket $g_i(v)$, where $i = 1, \ldots , L$.

2) Query procedure: When processing a query point q, search buckets $g_1(q), \ldots , g_L(q)$ and get all points $v_1, \ldots , v_n$ in these buckets as candidate ANNs. For each v_j, $j = 1, \ldots , n$, if $D(q , v_j) \leq r$, return v_j.

C. Locality Sensitive Hashing Based on p-Stable Distributions

[4] proposed LSH families based on p-stable distributions where each hash function is defined as:

$$h(v) = \left\lfloor \frac{a \cdot v + b}{w} \right\rfloor, \tag{1}$$

where a is a d-dimensional vector with entries chosen independently from a p-stable distribution and b is a real number chosen uniformly from the range $[0, w]$.

For any two vectors v_1, v_2 in the data domain, let $x = \left\| v_1 - v_2 \right\|_p$, and f_p is the probability density function of the absolute value of the p-stable distribution, we can prove that

$$p(x) = \Pr\left[h(v_1) = h(v_2)\right] = \int_0^w \frac{1}{x} f_p(\frac{t}{x})(1 - \frac{t}{w})dt. \tag{2}$$

For a fixed parameter w, the probability of collision $p(x)$ decreases monotonically with x, so $h(v)$ belongs to LSH family.

Exact Euclidean LSH (E2LSH) is a popular implementation of LSH based on p-stable distributions using generic LSH scheme which can be used in Euclidean space, and we use it to compare with our method in this paper.

III. Analysis of Locality Sensitive Hashing

We will give a brief analysis of generic LSH scheme in this section. Before that, let's define a function $t(x)$ first, where x denotes the distance between query point q and any point v in the data domain, and $t(x)$ denotes the probability of that v is the candidate ANN of q. In order to solve (r, δ)-NN problem effectively, this function should have two properties:

- if $x \leq r$ then $t(x) \geq \delta$,
- if $x \geq (1+c)r$ then $t(x) \leq \lambda$ (λ is small for large c).

The first property guarantees that each point v satisfying $D(q, v) \leq r$ should be returned with a probability at least δ while the second property guarantees that the candidate ANN set contains only a small number of false ANNs. The smaller λ is, the fewer comparisons will be made.

It is easy to find a function $h(v)$ belonging to family H in some degree while it is hard to meet the above two conditions at the same time by using a single $h(v)$ as hash function for the only hash table, and that's the reason why generic LSH scheme was proposed. For generic LSH scheme, it is easy to see

$$t(x) = f(p(x)^k) = 1 - (1 - p(x)^k)^L , \tag{3}$$

where $p(x)$ denotes the probability of that $h(q)$ equals $h(v)$. We know that $f(p(x)^k)$ increases monotonically with $p(x)^k$ when $p(x)^k$ is in the interval close to 0, and the larger L is, the more rapidly $f(p(x)^k)$ increases. Since $p(x)$ may drop slowly when $p(x)$ is small, $h(v)$ is not suitable to be used as the hash function of a hash table. To solve this problem, $g(v)$ is used. Compared with just using $h(v)$ as hash function of a hash table, using $g(v)$ insures that $p(x)^k$ drops rapidly with x when $p(x)^k$ is small. Thus, it is obvious to see $t(x)$ holds the two properties and this scheme can be used to solve (r, δ)-NN problem.

From the analysis above, we can see that to make LSH work better, L should be large, but a lot of memory is required at the same time. What's more, generic LSH scheme guarantees that any function $h(v)$ belonging to LSH family can be used to solve (r, δ)-NN problem. But it does not make full use of the properties of $h(v)$ in some degree. So we make a further study on the properties of $h(v)$ based on p-stable distributions and propose an improved LSH scheme which can be used for solving (r, δ)-NN problem better.

IV. Frequency Based Locality Sensitive Hashing

A. Frequency Based Locality Sensitive Hashing Scheme

Our new scheme is similar to generic LSH scheme except that it uses a single $h(v)$ as hash function and sets a frequency threshold m to select those points which collide with query point more than m times as candidate ANNs. It works as follows:

1) Index construction: For an integer L ($L > 1$), choose L functions $h_1, \ldots , h_L$ from H (based on p-stable distributions), independently and uniformly at random. For any point v in the input dataset, store it in the bucket $h_i(v)$, where $i = 1, \ldots , L$.

2) Query procedure: For a query point q, search buckets $h_1(q), \ldots , h_L(q)$ and get all points $v_1, \ldots , v_n$ which appear no less than m times in these buckets as candidate ANNs. For each v_j, $j = 1, \ldots , n$, if $D(q , v_j) \leq r$, return v_j.

B. *Correctness and Computational Complexity Analysis*

From our algorithm above, it is obvious to see that

$$t(x) = f(p(x)) = \sum_{i=m}^{L} C_L^i p(x)^i (1-p(x))^{L-i}\,, \tag{4}$$

where x denotes the distance between query point q and any point v in the data domain, $t(x)$ denotes the probability of that v is the candidate ANN of q, and $p(x)$ denotes the probability of that q collides with v in a hash table. Let $u = p(x)$, then

$$f(u) = \sum_{i=m}^{L} C_L^i u^i (1-u)^{L-i}\,. \tag{5}$$

It is easy to see $f(u)$ is a monotonically increasing function since

$$f'(u) = LC_{L-1}^{m-1} u^{m-1}(1-u)^{L-m} > 0\,. \tag{6}$$

We can get that $f'(u)$ is monotonically increasing in the interval (0, θ), and is monotonically decreasing in (θ, 1) while reaching its maximum when $u = \theta$ ($\theta = (m-1)/(L-1)$). Moreover, there is an interval (0, α) in which $f'(u)$ is close to 0; correspondingly, there exists an interval (β, 1) where $f'(u)$ is close to 0, too. It is easy to see that $f(0)=0$ and $\lim_{u\to 1} f(u) = 1$. So $f(u)$ is close to 0 and increases slowly when u is in the interval (0, α) while close to 1 and increasing slowly with u in the interval (β, 1). Finally, we can get such a conclusion that $f(u)$ increases rapidly in the interval [α, β] containing θ. Because θ increases as m increases, the interval [α, β] shifts to right with m increasing. That is to say, we can change the interval in which $f(u)$ increases rapidly by changing the value of m, as shown in Fig. 1.

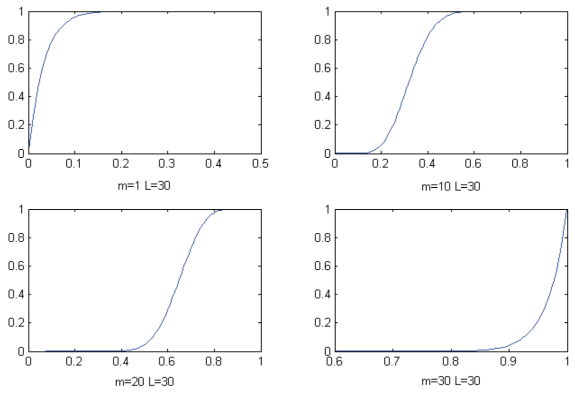

Figure 1. Fuction images of $f(u)$ with different m

A simple calculation from (2) shows that

$$p(x) = 1 - erfc(\frac{w}{\sqrt{2}x}) - \frac{2x}{\sqrt{2\pi}w}(1 - e^{-\frac{w^2}{2x^2}})\,. \tag{7}$$

It is difficult to study the properties of $p(x)$ according to its function expression, so we draw the function image (Fig. 2). Here we can assume that $r = 1$, since otherwise, we can scale down all the points by a factor of r. The range of x is set to [1, 2]. We observe that $p(x)$ increases with the value of w. When w is in the interval (0, 2), $p(x)$ is small and drops slowly in the interval [1, 2); similarly, when w is in (5, 10), $p(x)$ is large and drops slowly in [1, 2), too; but when w is in the interval [2, 5], $p(x)$ drops rapidly in [1, 2).

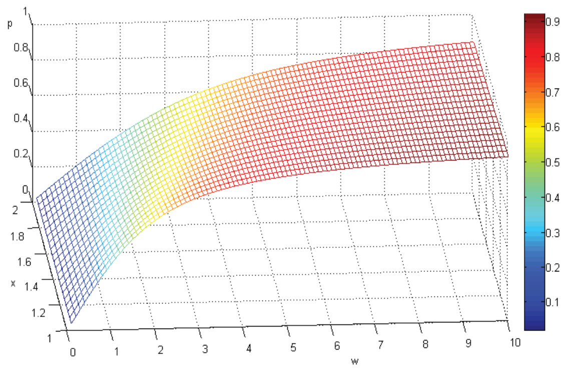

Figure 2. Fuction image of $p(x)$

It is for the reason that $p(x)$ drops slowly in the interval [r, (1+c)r) when $p(x)$ is small that we can not use a single $h(v)$ as the hash function of a hash table but use $g(v)$ instead in generic LSH scheme. But if we set w in the interval [2, 5], and set m properly so that the interval [α, β] intersects with [p((1+c)r), $p(r)$], using just a single $h(v)$ as the hash function is enough to solve (r, δ)-NN problem effectively since $t(x)$ decreases monotonically with x and drops rapidly as x increases in the interval [r, (1+c)r).

For E2LSH, let $u = p(x)^k$ and $L = 30$ in (3), the function image of $f(u)$ is the one with $m = 1$ in Fig. 1, and it increases rapidly with u close to 0. But $f(u)$ is close to 0 and increases slowly when u is close to 0 for FBLSH, as mentioned earlier. So for a large c, t((1+c)r) in FBLSH is smaller. In fact, there is a threshold l, for any $L > l$, FBLSH makes sure that t((1+c)r) is much smaller with the same $t(r)$ compared with E2LSH, and the difference becomes relatively larger with L increasing. The mathematics proof is complicated, so we prove it using Matlab. When w is 4, l is about 20. Actually, E2LSH has poor performance when L is less than 20, and it needs hundreds of hash tables to reach the same quality as FBLSH when FBLSH uses only dozens of hash tables. In other words, FBLSH needs fewer hash tables and comparisons than E2LSH with the same accuracy. Besides, FBLSH needs only about 4nL bytes memory while E2LSH needs about 12nL bytes, where n is the size of dataset. That's why FBLSH can reduce the space cost effectively.

The time for computing distances in FBLSH is smaller than that in E2LSH since FBLSH needs fewer comparisons than E2LSH. In E2LSH, $g(v)$ is the concatenation of k LSH functions, to reduce the memory needed for storing hash values, another two hash computations are needed besides k hash computations of $g(v)$, which is time-consuming, especially when L is very large. For small datasets, the frequency counting time is much less than the time for getting candidate

ANNs in E2LSH, so FBLSH consumes much less query time than E2LSH. For large datasets, the frequency counting time becomes longer and FBLSH has similar query time with E2LSH.

V. Experiments

In this section, we will describe the configurations of our experiments and show that our approach outperforms E2LSH through experimental results.

A. Experimental Setup

1) Implementation details: FBLSH is easy to implement in many ways. In our implementation, we set w in the interval [2, 5] and choose L which results in best query time. The parameters of E2LSH are set to the optimal ones estimated by itself. δ is set to 0.9 for both two methods.

2) Datasets: Our experiments are based on two datasets, one is 128-dimensional SIFT feature vectors of size 1M, the other is 1M 960-dimensional GIST feature vectors. Both of them are from INRIA ANN datasets for evaluation of ANN search algorithms. For each dataset, we generate several subsets with different size to perform experiments on.

3) Evaluation benchmark: For each dataset, we picked a set of 100 points from query set as queries, and we set the search radius properly so that the average number of near neighbors is 100. (For 10K datasets, the number is 20.) Since precision does not need to be considered, we mainly compare recall, query time, and memory used. The query time is measured by averaging the query time over 100 queries, and the memory used does not contain the memory for storing the dataset. Each experiment is repeated 5 times and the average is recorded.

4) Hardware and software: The evaluation is done on a PC with 4 32-bit 2.66GHz CPUs and 4GB RAM (about 3GB available). The operating system is linux with kernel 2.6.32.

B. Experimental Results

Table 1 and Table 2 show the average results of E2LSH and FBLSH methods with datasets of different size, and we compare the two methods from recall, query time, and memory used.

1) Recall: Each method has a higher recall compared with the given δ, and the recall of FBLSH is slightly higher than that of E2LSH.

2) Time: FBLSH outperforms E2LSH for small datasets and has similar time cost for large ones since the frequency counting time is increasing with the size of dataset. The higher the dimension is, the more time for computing distances our method saves, and the better our method performs. When the dataset is large enough to the capacity of main memory, 750K GIST dataset for example, E2LSH acts as bad as linear search while our method still needs only a little time.

3) Space: When memory is enough for E2LSH, FBLSH can reduce the extra memory used by a factor of 56 to 106, even for datasets larger than 500K, our approach can still reduce the memory by an order of magnitude.

VI. Conclusion

In this paper, we proposed a new LSH scheme called FBLSH which can reduce the extra space cost by several orders of magnitude with less (or similar) time cost while achieving better search quality compared with E2LSH. We are now trying to reduce the query time further and expect some improvements of LSH to be done based on our scheme.

TABLE I. Search Performance Comparison (SIFT)

Size	Method	Recall	Time (s)	L	Space Ratio
10K	E2LSH	0.951	0.00065	1485	97
	FBLSH	0.965	0.00042	46	1
100K	E2LSH	0.948	0.00314	1485	106
	FBLSH	0.950	0.00347	42	1
1M	E2LSH	0.959	0.02867	153	16
	FBLSH	0.983	0.03880	28	1

TABLE II. Search Performance Comparison (GIST)

Size	Method	Recall	Time (s)	L	Space Ratio
10K	E2LSH	0.937	0.00220	1485	56
	FBLSH	0.949	0.00082	79	1
100K	E2LSH	0.981	0.00420	1485	84
	FBLSH	0.984	0.00317	53	1
500K	E2LSH	0.923	0.01846	153	14
	FBLSH	0.982	0.01277	33	1
750K	E2LSH	0.982	0.72324	6	1
	FBLSH	0.981	0.03041	30	1.7

References

[1] Bentley and J. L. , “Multidimensional binary search trees used for associative searching,” Communications of the ACM, vol. 18, no. 9, pp. 509–517, 1975.

[2] C. B¨ohm, S. Berchtold, and D. Keim, “Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases,” ACM Computing Surveys, vol. 33, no. 3, pp. 322–373, 2001.

[3] P. Indyk and R. Motwani, “Approximate nearest neighbors: towards removing the curse of dimensionality,” in Proc. of Symposium on Theory of Computing, 1998, pp. 604–613.

[4] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni, “Locality-sensitive hashing scheme based on p-stable distributions,” in Proc. of annual symposium on Computational geometry, 2004, pp. 253–262.

[5] A. Andoni and P. Indyk, “Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions,” Communications of the ACM, vol. 51, no. 1, 2008.

[6] H. Jegou, L. Amsaleg, C. Schmid, and P. Gros, “Query-adaptative locality sensitive hashing,” in Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing, 2008, pp. 825.

[7] R. Panigrahy, “Entropy based nearest neighbor search in high dimensions,” in Proc. of annual ACM-SIAM symposium on Discrete algorithm, 2006, pp. 1186–1195.

[8] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, “Multi-probe LSH: efficient indexing for high-dimensional similarity search,” in Proc. of International Conference on Very Large Data Bases, 2007, pp. 253–262.

[9] A. Joly and O. Buisson, “A posteriori multi-probe locality sensitive hashing,” in Proc. of ACM International Conference on Multimedia, 2008.