Neighborhood Voting: A Novel Search Scheme for Hashing

Yan Xiao, Jiafeng Guo, Yanyan Lan, Jun Xu and Xueqi Cheng
University of Chinese Academy of Sciences, Beijing, China
CAS Key Lab of Network Data Science and Technology,
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
xiaoyanict@foxmail.com,{guojiafeng,lanyanyan,junxu,cxq}@ict.ac.cn

ABSTRACT

Hashing techniques for approximate nearest neighbor search (ANNS) encode data points into a set of short binary codes, while trying to preserve the neighborhood structure of the original data as much as possible. With the binary codes, the task of ANNS can be easily conducted over large-scale dataset, due to the high efficiency of pairwise comparison with the Hamming distance. Although binary codes have low computation and storage cost, the data are heavily compressed so that partial neighborhood structure information would be inevitably lost. To address this issue, we propose to introduce the k-nearest neighbors (k-NNs) in the original space into the Hamming space (i.e., associating a binary code with its original k-NNs) to enhance the effectiveness of existing hashing techniques with little overhead. Based on this idea, we develop a novel search scheme for hashing techniques namely neighborhood voting, i.e., each point retrieved by a query code will vote for its neighbors and itself, and the more voted, the better candidates. In this way, search in hashing is not simply the collision between codes (i.e., query code and candidate code), but also the collision between neighbors (i.e., neighbors of candidate points). The underlying assumption is that the true neighbors of a query point should be close to each other, while points with similar binary codes but seldom be the neighbors of other candidate points would be false positives. We introduce a novel data structure called aggregated hash table for implementing our idea and accelerating the online search process. Experimental results show that our search scheme can significantly improve the search effectiveness while having good efficiency over different existing hashing techniques.

CCS CONCEPTS

• **Information systems → Search engine indexing**;

KEYWORDS

Nearest Neighbor Search; Hashing; Information Retrieval

ACM Reference Format:
Yan Xiao, Jiafeng Guo, Yanyan Lan, Jun Xu and Xueqi Cheng. 2018. Neighborhood Voting: A Novel Search Scheme for Hashing. In *The 27th ACM International Conference on Information and Knowledge Management (CIKM '18), October 22–26, 2018, Torino, Italy.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3269206.3269240

CIKM '18, October 22–26, 2018, Torino, Italy

ACM ISBN 978-1-4503-6014-2/18/10…$15.00
https://doi.org/10.1145/3269206.3269240

1 INTRODUCTION

Approximate nearest neighbor search is considered as a fundamental problem in many fields, such as image retrieval and data mining. One of the major approach for ANNS is hashing technique [2, 4, 9], whose main idea is to transform data points into compact binary hash codes. Searching with hash codes can be extremely efficient due to the low distance computation cost in the Hamming space. Moreover, the compact representation makes it scalable to large scale dataset due to the low cost of storage [9]. Most research efforts on hashing focus on the design of good hash functions, which aim to generate binary codes with shorter bits but better preserve the data closeness relationship in the original space. For example, locality sensitive hashing [4] employs a number of data-independent random projections as the hash function, and learning to hash approaches [2, 9] are proposed to learn the hash function from the data distribution. While good search efficiency can be achieved with hash codes, good search effectiveness is hard to be guaranteed since original data are heavily compressed so that partial neighborhood structure information would be inevitably lost.

Meanwhile, the neighborhood structure is well preserved in the graph-based ANNS methods [3, 5, 6], which employ a neighborhood graph as the index, e.g., k-nearest neighbor graph. However, graph-based methods search in the original space which may not be easily scalable to large scale dataset. The original high dimensional representations can be too large to be loaded into memory, resulting in additional expensive disk I/O cost [10].

Inspired by the graph-based methods, we propose to introduce the k-NNs in the original space into the Hamming space to enhance the effectiveness of the existing hashing techniques with little overhead. We associate each binary hash code with its original k-NNs, and develop a novel search scheme namely neighborhood voting, i.e., each point retrieved by a query code will vote for its neighbors and the more voted points will be considered as the better candidates. The underlying assumption is that the true neighbors of a query point should be close to each other, while points with similar binary codes that are rarely the neighbors of other candidate points would be false positives. In this way, we consider not only the collision between the query code and the candidate code, but also the collision between neighbors. This may bring back the points with large Hamming distance that are actually close, and discard the points with small Hamming distance that are actually far away.

As described above, our method contains an additional voting process. To reduce the cost of voting, we propose to aggregate the votes from points with the same hash code and improve cache

hit rate. We further propose a novel data structure, called aggregated hash table, to implement this technology by incorporating the neighborhood structure into the traditional hash table.

We conduct experiments on public datasets to demonstrate the effectiveness and efficiency of the proposed search scheme implemented with the aggregated hash table. The empirical results show that our approach can significantly improve the search effectiveness while having good efficiency over different existing hashing techniques. Meanwhile, we do not need to compute in original space and the additional memory cost is negligible.

2 RELATED WORK

Hashing methods can be classified as locality sensitive hashing (LSH) or learning to hash [10] , based on whether data distribution is taken into consideration. LSH adopts a number of random projections as single hash functions and concatenates them as a compound hash function [4]. Learning to hash has been a popular topic in recent literature due to that the hash function is learned from the data distribution, which can achieve better search effectiveness. For example, iterative quantization (ITQ) [2] learns a rotation of zero-centered data to map data to a binary hypercube.

The direct search with hash codes is to perform an exhaustive search of the codes in database using bit operations. Since hash codes can be directly used as indices, hash table lookup [10] can be adopted to speed up the search in Hamming space. Points with the same hash code are stored in a bucket indexed by the hash code, and all the buckets are organized as a hash table. Searching with a hash table can be performed by looking up the buckets that are within small Hamming distance of the query code, and indices of these buckets can be directly obtained via flipping a few bits of the query code. Hash table lookup avoids the exhaustive search and works well for codes within 32 bits, but lose its efficiency for longer bits since the combination number of the flipped bits grows exponentially with the search radius [8]. Other efforts propose to use an extra structure for ANNS in the Hamming space, such as hierarchical k-means trees [7] and multi-index hashing [8].

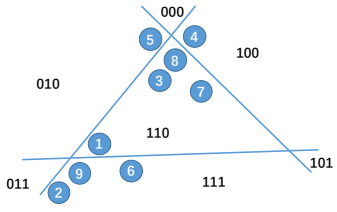
3 OUR APPROACH

3.1 Key Idea

Although binary codes have low computation and storage cost, partial neighborhood structure information is inevitably lost due to the compact representation. To address this issue, we introduce the k-NNs in the original space into the Hamming space and develop the neighborhood voting scheme to boost hashing techniques. Specifically, each point retrieved will vote for its k-NNs, and this point will also have one vote due to its collision with the query. The points with highest number of votes can be returned as candidates when a certain number of points have voted.

Our approach comes from the assumption that the true neighbors of a query point should be close to each other, while points with similar binary codes that are rarely the neighbors of other candidates could be false positives. Meanwhile, the hashing method is expected to recall a certain number of true neighbors. Therefore, these true neighbors will vote for each other, thus getting more votes than outliers. In this way, the missed true neighbors in hash could be taken back while outliers could be discarded.

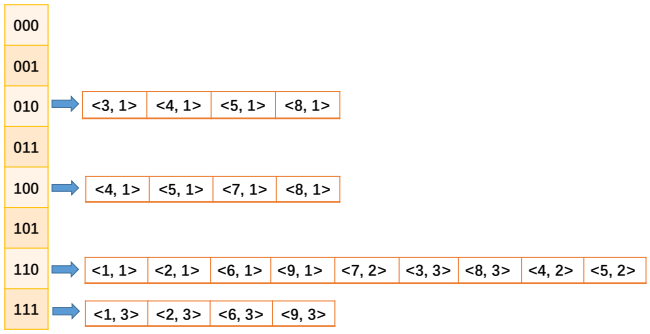
3.2 Implementation



(a) 3-bit LSH. The number in each blue point denotes its unique identifier (ID). The binary sequence bits denote the hash code of corresponding area, e.g., the binary code of point 4 is 100.

1	9	6	2
2	9	1	6
3	8	5	7
4	8	5	7
5	8	4	3
6	1	9	2
7	3	8	4
8	3	4	5
9	2	1	6

(b) 3-NN graph. The first column is the ID of each point to which 3-NN list are attached, e.g., the 3-nearest neighbors of point 7 is point 3, 8 and 4.



(c) Aggregated hash table. Each bucket stores the <point ID, voted number> pairs generated by aggregating all the votes from the points in this bucket. For example, in the bucket whose hash code is 010, point 5 has one vote initially and votes for its 3-NNs: point 8, 4 and 3, resulting in these points all having one vote.

Figure 1: Illustration for the aggregated hash table.

Here we use an example to describe of our approach. As shown in Figure 1(a), we have a few data points which can be mapped to binary codes using LSH. To record the neighbors of each point, we approximately and efficiently construct a k-NN graph by NN-Descent [1], where k denotes the number of neighbors. As shown in Figure 1(b), the k-NN graph are organized by attaching a k-NN list to each point. The search process of our approach follows the same framework with hash table lookup. Differently, the points in visited buckets are not immediately considered as the neighbor candidates but will vote for its neighbors instead. To avoid the cost of ranking all the votes, the points whose votes exceed a pre-defined threshold will be considered as good candidates and the search process stops when enough candidates are retrieved.

This straightforward implementation can enhance the effectiveness of hashing methods. We further have two considerations about the efficiency. Firstly, votes from points of one bucket can be aggregated in the offline stage to accelerate the online voting process. Secondly, points in one bucket will look up the k-NN graph frequently, which is not cache-friendly because of the approximately random distribution of the entries in the graph.

Based on these two considerations, we propose the aggregated hash table as illustrated in Figure 1(c). Each bucket in the aggregated hash table stores the <point, voted number> pair list, where these pairs are generated by aggregating all the votes from points in the same bucket. Therefore, we can collect votes bucket by bucket rather than point by point. Meanwhile, the k-NN graph has been incorporated into the aggregated hash table and can be discarded during the online search stage. The detailed construction algorithm of the aggregated hash table is formally described in Algorithm 1, while corresponding search algorithm is described in Algorithm 2.

3.3 Complexity Analysis

The time complexity in offline stage consists of the k-NN graph construction and the aggregated hash table construction. NN-Descent has a low empirical complexity of $O(N^{1.14})$ [1], where N is the total number of points. As for the aggregated hash table, the hash codes of N points are processed in one pass. Since the neighborhood lookup is O(1) time complexity while the vote counting is also O(1), the voting process of one point has $O(k)$ time complexity. Therefore, the total time complexity of a aggregated hash table construction is $O(N^{1.14} + kN)$. In the online search stage, our approach has the same encoding process (i.e., encode the query to a hash code) with hash table lookup and contains an additional voting process, which will increase the search cost. On the other hand, our approach introduces neighbor points and less buckets need to be visited, which will decrease the search cost. The entire search efficiency depends on the trade-off between the additional voting cost and the visiting reduction. We will give the empirical result in our experiments.

As for the memory, without the voting aggregation, the additional memory cost of the k-NN graph is $O(kN)$. With the aggregated hash table, the worst case is that no two points have the same hash code. In this case, each bucket stores additional k neighbors and voted numbers of all the points. Thus the additional memory complexity of our approach is still $O(kN)$ in the worst case.

Algorithm 1 Construction algorithm for the aggregated hash table

Input: k-NN graph G, hash codes of N points $\{c_j\}_{j\in[N]}$
Output: Aggregated hash table A

```
for j = 1 to N do
    if c_j ∉ A then
        A[c_j] ← {}
    end if
    for v ∈ {j} ∪ G[j] do
        if v ∉ A[c_j] then
            A[c_j][v] ← 0
        end if
        A[c_j][v] ← A[c_j][v] + 1
    end for
end for
return A
```

3.4 Discussions

As new points come, the neighbors may change. Rather than rebuilding the data structure, we could adopt an approximate updating strategy. Specifically, we perform ANNS to find the k-NNs for a new point and update the neighbors of these k-NNs with the new point. With the aggregated hash table, only buckets where the new point and its k-NNs fall into will be updated.

Algorithm 2 Search algorithm with the aggregated hash table

Input: Query point q, candidate number n, aggregated hash table A and corresponding l-bit hash function H, voting threshold m
Output: n neighbor candidates

```
C ← array of size N whose entries are initialized to 0
S ← ∅
for r = 0 to l do
    B ← list of all buckets with r Hamming distance to H(q) in A
    for b ∈ B do
        for v ∈ A[b] do
            C[v] ← C[v] + A[b][v]
            if C[v] ≥ m then
                S ← S ∪ {v}
                if #S ≥ n then
                    return S
                end if
            end if
        end for
    end for
end for
```

Note that we introduce our idea based on hash table lookup since this is an efficient way with compact short hash codes. Due to the fact that the input of our approach is a sequence of retrieved points and the output is a better sequence, our proposed neighborhood voting search scheme can be naturally applied over other search schemes, such as other binary indices [7, 8] for long hash codes. Moreover, our proposed voting aggregation technology can also be implemented beyond the hash table. For example, the leaf of a hierarchical k-means tree [7] is actually similar to one bucket of a hash table, and votes from points fall in one leaf can be aggregated in offline stage to accelerate online search. We leave the empirical test of our approach over other search schemes as further study.

4 EXPERIMENTS

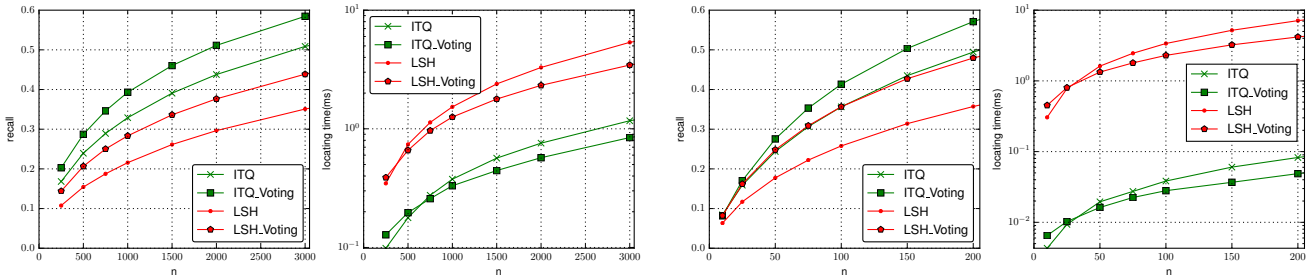
4.1 Experimental Settings

The experiments are conducted on a machine with 2.7 GHz Intel Core i5-5257U CPU and 8 GB memory. Hash codes of different methods are generated by matlab code, while the search methods are implemented with C++ and compiled with *g++6* and *O3* option. A single thread is used in testing stage.

The evaluation metric of the effectiveness is averaged recall@n in finding 10-NNs. As for the efficiency, we use averaged locating time (i.e., the time to locate these returned points) since our approach has the same query encoding time with boosted hashing methods.

We conduct experiments with the representative data-independent locality sensitive hashing (LSH) and data-dependent iterative quantization (ITQ) on two popular public datasets: SIFT-1M[1] and MNIST[2] to evaluate the effectiveness and efficiency of our approach. We reshape the 28 x 28 grayscale images in MNIST into vectors and

[1] http://corpus-texmex.irisa.fr/
[2] http://yann.lecun.com/exdb/mnist/



(a) LSH and ITQ on SIFT-1M (b) LSH and ITQ on MNIST

Figure 2: Results on SIFT-1M and MNIST with different base hashing methods.

the detailed information of the datasets are summarized in Table 1. 32-bit hash codes and 10-NN graph are used for all the hashing methods. Constructing the graph costs 293s on SIFT-1M and 23s on MNIST respectively using 4 threads in the offline stage.

Table 1: Detailed information of the datasets

Dataset	Dimension	Base number	Query number
SIFT-1M	128	1,000,000	10,000
MNIST	784	60,000	10,000

4.2 Experimental Results

The key parameter is voting threshold (i.e., m). We set $m = 0$ to denote the boosted hashing method and the impact of m based on ITQ is shown in Table 2. We can see $m = 2$ can significantly improve the effectiveness, thus being used as our default setting. Although $m \geq 3$ can further improve the results, it will decrease efficiency at the same time. Note that when $m = 1$, our voting scheme reduces to one-iteration neighborhood expansion [5] . The results show that simple neighborhood expansion is not robust, as it may introduce noise and decrease the effectiveness on some dataset.

Table 2: Recall@100 based on ITQ with different m

m	0	1	2	3
SIFT-1M	0.101	0.090	0.119	0.129
MNIST	0.356	0.360	0.413	0.417

As we can see in Figure 2, on both datasets, our approach can boost corresponding hashing methods significantly and consistently in terms of different recall@n, which demonstrates the effectiveness of our approach. We also find that our approach usually reduces the locating time with the same number of candidates, leading to good search efficiency. For example, on SIFT-1M, our approach can improve the recall@1000 of ITQ from 0.329 to 0.393. Meanwhile, the locating time decreases from 0.376ms to 0.258ms. Note that under very small n, the location time might increase a little. In this case, the visiting reduction will be smaller than the additional voting cost in our approach, leading to a little bit higher locating time.

The memory overhead of our approach is small as compared with the size of original dataset (i.e., 488M for SIFT-1M and 179M for MNIST). For example, with the aggregated hash table based on ITQ, the additional overhead is 63M on SIFT-1M and 3.7M on MNIST in the case of $k = 10$.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we introduce k-NNs into hashing techniques and develop a novel search scheme namely neighborhood voting to enhance the search effectiveness of existing hashing methods. We also propose a novel data structure called aggregated hash table for efficient implementation. Experimental results show that our search scheme can significantly improve the search effectiveness while having good efficiency over different existing hashing techniques. In the future, we want to test our idea over other search schemes and investigate whether some theoretical guarantees can be provided.

6 ACKNOWLEDGMENTS

This work was funded by the 973 Program of China under Grant No. 2014CB340401, the National Natural Science Foundation of China (NSFC) under Grants No. 61425016, 61472401, 61722211, 61872338 and 20180290, the Youth Innovation Promotion Association CAS under Grants No. 20144310 and 2016102, and the National Key R&D Program of China under Grants No. 2016QY02D0405.

REFERENCES

[1] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*. ACM, 577–586.
[2] Yunchao Gong, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. 2013. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *TPAMI* 35, 12 (2013), 2916–2929.
[3] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. 2011. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI*, Vol. 22. 1312.
[4] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*. ACM, 604–613.
[5] Zhongming Jin, Debing Zhang, Yao Hu, Shiding Lin, Deng Cai, and Xiaofei He. 2014. Fast and accurate hashing via iterative nearest neighbors expansion. *IEEE transactions on cybernetics* 44, 11 (2014), 2167–2177.
[6] Yu A Malkov and Dmitry A Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *arXiv preprint arXiv:1603.09320* (2016).
[7] Marius Muja and David G Lowe. 2012. Fast matching of binary features. In *CRV*. IEEE, 404–410.
[8] Mohammad Norouzi, Ali Punjani, and David J Fleet. 2012. Fast search in hamming space with multi-index hashing. In *CVPR*. IEEE, 3108–3115.
[9] Antonio Torralba, Rob Fergus, and Yair Weiss. 2008. Small codes and large image databases for recognition. In *CVPR*. IEEE, 1–8.
[10] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. 2014. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927* (2014).