Fast Nearest Neighbor Search in the Hamming Space

Zhansheng Jiang[1(✉)], Lingxi Xie[2], Xiaotie Deng[1], Weiwei Xu[3], and Jingdong Wang[4]

[1] Shanghai Jiao Tong University, Shanghai, People's Republic of China
jzhsh1735@sjtu.edu.cn, deng-xt@cs.sjtu.edu.cn
[2] Tsinghua University, Beijing, People's Republic of China
198808xc@gmail.com
[3] Hangzhou Normal University, Hangzhou, People's Republic of China
weiwei.xu.g@gmail.com
[4] Microsoft Research, Beijing, People's Republic of China
jingdw@microsoft.com

Abstract. Recent years have witnessed growing interests in computing compact binary codes and binary visual descriptors to alleviate the heavy computational costs in large-scale visual research. However, it is still computationally expensive to linearly scan the large-scale databases for nearest neighbor (NN) search. In [15], a new approximate NN search algorithm is presented. With the concept of *bridge vectors* which correspond to the cluster centers in Product Quantization [10] and the *augmented neighborhood graph*, it is possible to adopt an *extract-on-demand* strategy on the online querying stage to search with priority. This paper generalizes the algorithm to the Hamming space with an alternative version of k-means clustering. Despite the simplicity, our approach achieves competitive performance compared to the state-of-the-art methods, *i.e.*, MIH and FLANN, in the aspects of search precision, accessed data volume and average querying time.

Keywords: Approximate nearest neighbor search · Hamming space · Bridge vectors · Augmented neighborhood graph

1 Introduction

Compact binary codes [7,8] and binary visual descriptors [4–6] play a significant role in computer vision applications. Binary vectors have a lot of advantages, including the cheap storage cost and the low consumption in computing the distance between binary vectors through a bitwise XOR operation. Despite the efficiency of binary vector operations, it is often time-consuming to linearly scan a large-scale database, seeking for the nearest neighbor.

This work was done when Zhansheng Jiang was an intern at Microsoft Research, P.R. China.

Q. Tian et al. (Eds.): MMM 2016, Part I, LNCS 9516, pp. 325–336, 2016.
DOI: 10.1007/978-3-319-27671-7_27

A lot of efforts are made in accelerating nearest neighbor search. MIH [2] is an exact nearest neighbor search algorithm. However, the index construction and the query process for large-scale search databases are very time-consuming and therefore impractical. FLANN [3] is an approximate nearest neighbor search method, the precision of which is often low in the scenarios of long binary codes or large-scale search databases, even with a large number of accessed data points. In [15], the authors present the concept of *bridge vectors*, which are similar to the cluster centers in Product Quantization [10], and construct a *bridge graph* by connecting each bridge vector to its nearest vectors in the database. In the online querying procedure, it is possible to organize data in a priority queue and design an *extract-on-demand* strategy for efficient search.

Neighborhood graph search has attracted a lot of interests [12,15] because of the low cost in extracting neighbor vectors and the good search performance. The index structure is a directed graph connecting each vector to its nearest neighbors in a search database. The graph search procedure involves measuring the priority of each candidate. Accessed vectors are organized in a priority queue ranked by their distance to the query vector. The top vector in the priority queue is popped out one by one and its neighborhood vectors are pushed into the queue. This process continues until a fixed number of vectors are accessed.

In this paper, we generalize the algorithm in [15] to the Hamming space. The major novelty of this work lies in that we generalize the previous algorithm to the Hamming space with an alternative version of k-means. Experiments reveal that, despite the simplicity, our approach achieves superior performance to the state-of-the-art methods, *i.e.*, MIH and FLANN, in the aspects of search precision, accessed data volume and average querying time.

The rest of this paper is organized as follows. In Sect. 2, we review previous works of nearest neighbor search in the Hamming space. We introduce our approach in Sect. 3. Experiment results are presented in the next section. In the end, we summarize this paper and state the conclusions.

2 Related Works

In this section, we will introduce two previous works which are popular for nearest neighbor search in the Hamming space.

2.1 Multi-index Hashing

Multi-index hashing (MIH) algorithm is presented in [2] to achieve a fast NN search in the Hamming space. The algorithm indexes the database m times into m different hash tables, then during the search for each query, it collect NN candidates through looking up the hash tables by taking advantage of the Pigeonhole Principle.

In detail, each b-bit binary code $\mathbf{h}$ in the database is split into m disjoint substrings $\mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(m)}$, each of which is b/m in length, assuming that b is divisible by m. Then the binary code is indexed in m hash tables according to

its m substrings. The key idea rests on the following proposition: Given $\mathbf{h}, \mathbf{g}$, and $\|\mathbf{h} - \mathbf{g}\|_H \leq r$, where $\|\cdot\|_H$ denotes the Hamming norm, there exists k, $1 \leq k \leq m$, such that

$$\|\mathbf{h}^{(k)} - \mathbf{g}^{(k)}\| \leq \lfloor \frac{r}{m} \rfloor.$$

The proposition can be proved by the Pigeonhole Principle. During the search for query $\mathbf{q}$ with substrings $\{\mathbf{q}^{(i)}\}_{i=1}^{m}$, we collect from i^{th} hash table those entries which are within the Hamming distance $\lfloor \frac{r}{m} \rfloor$ to $\mathbf{q}^{(i)}$, denoted as $\mathcal{N}_i(\mathbf{q})$. Then the set $\mathcal{N} = \bigcup_{i=1}^{m} \mathcal{N}_i(\mathbf{q})$ will contain all binary codes within the Hamming distance r to $\mathbf{q}$.

The key idea stems from the fact that, in the case of n binary b-bit code and $2^b \gg n$, when we build a full Hash table, most of the buckets are empty. When we retrieve the r-neighbors of a query q, we need to traverse 2^r buckets, which is very costly when r is large. However, in this approach, many buckets are merged together by marginalizing over different dimensions of the Hamming space. As a result, the number of visited buckets is greatly reduced from 2^r to $m \cdot 2^{\lfloor \frac{r}{m} \rfloor}$. The downside is that not all the candidates in these merged buckets are the r-neighbors of the query, so we need to examine them one by one.

In this approach, the search cost depends on the number of visited buckets and the number of accessed candidates. As a result, a trade-off has to be made between them by choosing a proper m. When m is small, $\lfloor \frac{r}{m} \rfloor$ is large so we have to traverse many buckets. When m is large, many buckets are merged together so we have to check a large number of candidates one by one.

The major disadvantage of MIH algorithm lies in the heavy computational costs. When the code length b is large, *e.g.*, $b = 512$, either m or $\lfloor \frac{r}{m} \rfloor$ is too large so that the algorithm becomes less efficient, as shown in later experiments.

2.2 FLANN

FLANN [3] has been a well-know library for approximate nearest neighbor (ANN) search in the Euclidean space. In [3], the authors of FLANN introduce a method for ANN search in the Hamming space.

First, the algorithm performs a hierarchical decomposition of the Hamming space to build a tree structure. It starts with clustering all the data points into K clusters, where K is a parameter. The cluster centers are K points which are randomly selected and each data point is assigned to its nearest center in the Hamming distance. The decomposition is repeated recursively for each cluster until the number of points in a cluster is less than a threshold, in which case this cluster becomes a leaf node. The hierarchical decomposition of the database is repeated for several times and multiple hierarchical trees are constructed. The search performance significantly improves as the number of trees increases. The search process is performed by traversing multiple trees in parallel, which is presented in Algorithm 1 [3].

The approach of constructing multiple hierarchical trees is similar to k-d trees [13] or hierarchical k-means trees [14]. However, in these approaches, the strategy of building trees is well-optimized so the decomposition results are always

Algorithm 1. Searching Parallel Hierarchical Clustering Trees

Input hierarchical trees $\mathcal{T} = \{T_i\}$, query point $\mathbf{q}$
Output K nearest approximate neighbors of $\mathbf{q}$
Parameters the max number of examined points $L_{\max}$
Variables $\mathcal{PQ}$: the priority queue storing the unvisited branches; $\mathcal{R}$: the priority queue storing the examined data points; L: the number of examined data points.

$L \leftarrow 0$
$\mathcal{PQ} \leftarrow \emptyset$
$\mathcal{R} \leftarrow \emptyset$
for each tree T_i **do**
  call TraverseTree$(T_i, \mathcal{PQ}, \mathcal{R})$
end for
while $\mathcal{PQ} \neq \emptyset$ **and** $L < L_{\max}$ **do**
  $N \leftarrow$ top of $\mathcal{PQ}$
  call TraverseTree$(N, \mathcal{PQ}, \mathcal{R})$
end while
return K nearest points to $\mathbf{q}$ from $\mathcal{R}$

procedure TraverseTree$(T, \mathcal{PQ}, \mathcal{R})$

if T is a leaf node **then**
  examine all data points in T and add them in $\mathcal{R}$
  $L \leftarrow L + |T|$
else
  $\mathcal{C} \leftarrow$ child nodes of T
  $N_q \leftarrow$ nearest node to $\mathbf{q}$ in $\mathcal{C}$
  $\bar{\mathcal{C}} \leftarrow \mathcal{C} \setminus \{N_q\}$
  add all nodes in $\bar{\mathcal{C}}$ to $\mathcal{PQ}$
  call TraverseTree$(N_q, \mathcal{PQ}, \mathcal{R})$
end if

the same and constructing multiple trees is unnecessary. In this approach, since the K cluster centers are randomly selected and no further optimization is performed, multiple different trees can be constructed. The benefit of constructing multiple hierarchical trees is much more significant due to the fact that when explored in parallel, each tree will retrieve different candidates, thus the probability of finding the exact NNs is increased.

3 Our Approach

3.1 Data Structure

The major part of our data structure is nearly the same as in [15], but we will later generalize it to the Hamming space. The index structure consists of two components: the *bridge vectors* and the *augmented neighborhood graph*.

Bridge Vectors. Inspired by Product Quantization [10] and inverted multi-index [9], we propose to construct a set of bridge vectors, which are similar to

the cluster centers in Product Quantization. We split the vectors in the database into m dimensional chunks and cluster i-th dimensional chunk into n_i centers. The bridge vectors are defined as follows:

$$\mathcal{Y} = \times_{i=1}^{m} \mathcal{S}_i \triangleq \{\mathbf{y}_j = [\mathbf{y}_{j_1}^T \ \mathbf{y}_{j_2}^T \ \cdots \ \mathbf{y}_{j_m}^T]^T | \mathbf{y}_{j_i} \in \mathcal{S}_i\}.$$

where $\mathcal{S}_i$ is the center set for the i-th dimensional chunk. To be simplified and without loss of generality, we assume that $n_1 = n_2 = \cdots = n_m = n$. There is a nice property that finding the nearest neighbor over the bridge vectors is really efficient, which takes only $O(nd)$ in spite of the size of the set is n^m. Moreover, by adopting the Multi-sequence algorithm in [9], we can identify the second, the third and more neighbors each in $O(m^2 \log n)$ after sorting n_i centers in each dimensional chunk $\mathcal{S}_i$ in $O(mn \log n)$.

The major difference between this work and [15] lies in that we need to deal with the Hamming space, i.e., **the conventional Euclidean distance shall be replaced by the Hamming distance.** In this respect, we adopt an alternative version of k-means clustering for binary codes and the Hamming distance to obtain the center set for each dimensional chunk. We first initialize k cluster centers by randomly drawing k different data points from the database. Then we run two steps, the assignment step and the update step, iteratively. In the assignment step, we find the nearest center for each data point and assign the point to the corresponding cluster. In the update step, for each cluster, we find a Hamming vector which has a minimum average Hamming distance to all cluster members as the new cluster center. This is simply done by voting for each bit individually, *i.e.*, each bit of each cluster center takes the dominant case of this bit (0 or 1) assigned to this center.

Although our approach seems straightforward on the basis of k-means, it provides an opportunity to transplant other algorithms based on the Euclidean distance to the Hamming space. One of the major costs in k-means clustering lies in the distance computation, and the current version which computes the Hamming distance is much faster than the previous one which computes the Euclidean distance (5× faster). In the large-scale database, this property helps a lot in retrieving the desired results in reasonable time.

Augmented Neighborhood Graph. The augmented neighborhood graph is a combination of the neighborhood graph $\mathbf{G}$ over the search database vectors $\mathcal{X}$ and the bridge graph $\mathbf{B}$ between the bridge vectors $\mathcal{Y}$ and the search database vectors $\mathcal{X}$. The neighborhood graph $\mathbf{G}$ is a directed graph. Each node corresponds to a vector $\mathbf{x}_i$, and each node $\mathbf{x}_i$ is connected with a list of nodes that correspond to its neighbors.

In the bridge graph $\mathbf{B}$, each bridge vector $\mathbf{y}_j$ in $\mathcal{Y}$ is connected to its nearest vectors in $\mathcal{X}$. To avoid expensive computation cost, we build the bridge graph approximately by finding the top t (typically 1000 in our experiments) nearest bridge vectors for each search database vector through the Multi-sequence

Algorithm 2. Fast Nearest Neighbor Search in the Hamming Space (FNNS)

Input query point $\mathbf{q}$, bridge graph $\mathbf{B}$, neighborhood graph $\mathbf{G}$
Output K nearest approximate neighbors of $\mathbf{q}$
Parameters the max number of accessed points $L_{\max}$
Variables $\mathcal{PQ}$: the priority queue storing the accessed points; L: the number of accessed data points; $\mathbf{b}$: the current bridge vector.

```
1: b ← nearest bridge vector to q through the Multi-sequence algorithm in [9]
2: PQ ← {b}
3: L ← 0
4: while L < L_max do
5:    if b is top of PQ then
6:       call ExtractOnDemand(b, PQ)
7:    else
8:       t ← top of PQ
9:       examine and push neighbor points of t in neighborhood graph G to PQ
10:      L ← L + number of new points pushed to PQ
11:      pop t from PQ
12:   end if
13: end while
14: return K nearest points to q popped from PQ
```

procedure ExtractOnDemand($\mathbf{b}, \mathcal{PQ}$)

```
1: examine and push neighbor points of b in bridge graph B to PQ
2: L ← L + number of new points pushed to PQ
3: pop b from PQ
4: b ← next nearest bridge vector to q through the Multi-sequence algorithm in [9]
5: push b to PQ
```

algorithm in [9] and then keeping the top b (typically 50 in our experiments) nearest search database vectors for each bridge vector.

3.2 Search over the Augmented Neighborhood Graph

We first give a brief view of the search procedure in a neighborhood graph. In this procedure, we first select one or more seed points, and then run a priority search starting from these seed points. The priority search is very similar to the breadth-first search. In the breadth-first search, we organize all visited points in a queue, in which the points are sorted by the order that they are pushed into the queue. However, in the priority search, we organize the visited points in a priority queue, in which the points are sorted by the distance to the query point. In detail, we first push all seed points into the priority queue. During each iteration, we pop out the top element, which is the nearest to the query point, from the priority queue and push its unvisited neighbors in the neighborhood graph into the priority queue.

To exploit bridge vectors and the augmented neighborhood graph, we adopt an extract-on-demand strategy. At the beginning of the search, we push the bridge vector nearest to the query into the priority queue, and during the entire

search procedure, we maintain the priority queue such that it consists exactly one bridge vector. In each iteration, if the top element is a data point, the algorithm proceeds as usual; if the top element is a bridge vector, we extract its neighbors in the bridge graph and push the unvisited ones into the priority queue, and in addition we extract the next nearest bridge vector using the Multi-sequence algorithm [9] and push it into the priority queue. The algorithm ends when a fixed number of data points are accessed.

4 Experiments

4.1 Datasets and Settings

Datasets. We evaluate our algorithm on three datasets: 1 million BRIEF dataset, 1 million BRISK dataset and the 80 million tiny images dataset [1]. The BRIEF and BRISK datasets are composed of the BRIEF and BRISK features of 1 million Flickr images crawled from the Internet. BRIEF and BRISK features are 128-bit and 512-bit binary codes, respectively. For the 80 million tiny images dataset, we learn Hash codes based on GIST features [11] through LSH [8] and MLH [7] resulting in 512-bit binary codes. We carry on experiments in three different scales of search databases, *i.e.*, 1 M, 10 M and 79 M.

Evaluation. We precompute the exact k-NNs as ground truth based on the Hamming distance to each query in the test set. We use the precision to evaluate the search quality. For k-NN search, the precision is computed as the ratio of retrieved points that are the exact k nearest neighbors. We conduct experiments on multiple settings with the number of neighbors $k = 1$, $k = 10$ and $k = 50$.

We construct the test set by randomly choosing 10 K images excluded from the search database. We download the source codes of FLANN and MIH method and compare our approach with these two algorithms.

Settings. All the experiments are conducted on a single CPU core of server with 128 G memory. In our approach, binary codes are split into 4 dimensional chunks and each chunk is clustered into 50 centers. In FLANN, hierarchical clustering uses the following parameters: tree number 4, branching factor 32 and maximum leaf size 100.

4.2 Results

For FLANN and our approach, we report the search time and the precision with respect to the number of accessed data points. Since MIH is an exact search algorithm, the precision is always 1. We also report the best search time and the number of accessed data points by tuning the parameter m.

According to Table 1, FLANN takes about 50 % more search time than our approach to achieve the same precision. Meanwhile, our approach only accesses half of data points that FLANN accesses to achieve the same precision. For a larger k, the advantage of our approach is more significant. MIH has similar

Table 1. BRIEF 1 M Data

Algorithm	Volume	Time(ms)	Precision
$k = 1$			
FNNS	3000	0.350	0.974
	5000	0.569	0.984
	10000	**1.248**	**0.993**
FLANN	5000	0.508	0.947
	10000	0.939	0.975
	20000	**1.856**	**0.990**
MIH	21064	1.5	1.000
$k = 10$			
FNNS	3000	0.395	0.978
	5000	0.599	0.989
	10000	**1.342**	**0.996**
FLANN	5000	0.509	0.831
	10000	0.945	0.914
	20000	**1.831**	**0.964**
MIH	49655	3.6	1.000
$k = 50$			
FNNS	3000	0.517	0.971
	5000	0.838	0.985
	10000	**1.726**	**0.995**
FLANN	5000	0.522	0.693
	10000	0.963	0.840
	20000	**1.855**	**0.932**
MIH	72888	5.5	1.000

Table 2. BRISK 1 M Data

Algorithm	Volume	Time(ms)	Precision
$k = 1$			
FNNS	1000	0.122	0.755
	6000	0.865	0.971
	20000	**3.296**	**0.997**
FLANN	10000	1.566	0.859
	30000	4.578	0.959
	60000	**9.263**	**0.988**
MIH	776410	87.4	1.000
$k = 10$			
FNNS	1000	0.142	0.698
	6000	0.899	0.957
	20000	**3.517**	**0.995**
FLANN	10000	1.620	0.540
	30000	4.691	0.836
	60000	**9.508**	**0.948**
MIH	851400	97.6	1.000
$k = 50$			
FNNS	1000	0.231	0.612
	6000	1.281	0.932
	20000	**4.655**	**0.991**
FLANN	10000	1.641	0.234
	30000	4.758	0.633
	60000	**9.691**	**0.870**
MIH	891010	103.4	1.000

Table 3. LSH 1 M Data

Algorithm	Volume	Time(ms)	Precision
$k = 1$			
FNNS	6000	0.953	0.913
	20000	3.976	0.983
	50000	**11.504**	**0.996**
FLANN	30000	4.917	0.909
	50000	8.299	0.955
	100000	**17.083**	**0.990**
MIH	516118	84.1	1.000
$k = 10$			
FNNS	6000	0.989	0.894
	20000	4.159	0.978
	50000	**12.101**	**0.995**
FLANN	30000	5.000	0.724
	50000	8.331	0.850
	100000	**17.011**	**0.960**
MIH	596586	97.5	1.000
$k = 50$			
FNNS	6000	1.307	0.859
	20000	5.197	0.968
	50000	**14.547**	**0.992**
FLANN	30000	4.875	0.521
	50000	8.209	0.715
	100000	**16.906**	**0.917**
MIH	645739	105.3	1.000

Table 4. MLH 1 M Data

Algorithm	Volume	Time(ms)	Precision
$k = 1$			
FNNS	3000	0.551	0.971
	5000	0.981	0.989
	10000	**1.905**	**0.997**
FLANN	10000	1.506	0.905
	30000	4.413	0.980
	50000	**7.415**	**0.992**
MIH	302475	52.0	1.000
$k = 10$			
FNNS	3000	0.563	0.952
	5000	1.006	0.981
	10000	**1.947**	**0.995**
FLANN	10000	1.507	0.679
	30000	4.476	0.902
	50000	**7.479**	**0.960**
MIH	399496	69.9	1.000
$k = 50$			
FNNS	3000	0.747	0.920
	5000	1.295	0.964
	10000	**2.448**	**0.988**
FLANN	10000	1.534	0.415
	30000	4.382	0.774
	50000	**7.375**	**0.904**
MIH	472219	79.7	1.000

Table 5. LSH 10 M Data

Algorithm	Volume	Time(ms)	Precision
$k = 1$			
FNNS	20000	4.931	0.913
	50000	13.835	0.962
	100000	**31.880**	**0.980**
FLANN	80000	16.525	0.847
	100000	19.900	0.873
	200000	**57.020**	**0.937**
MIH	4302972	719.1	1.000
$k = 10$			
FNNS	20000	5.028	0.901
	50000	14.115	0.959
	100000	**32.456**	**0.981**
FLANN	80000	16.585	0.588
	100000	19.867	0.639
	200000	**59.723**	**0.799**
MIH	5320692	890.6	1.000
$k = 50$			
FNNS	20000	6.361	0.873
	50000	16.929	0.945
	100000	**37.501**	**0.974**
FLANN	80000	16.544	0.377
	100000	19.936	0.432
	200000	**65.875**	**0.648**
MIH	5774213	965.6	1.000

Table 6. MLH 10 M Data

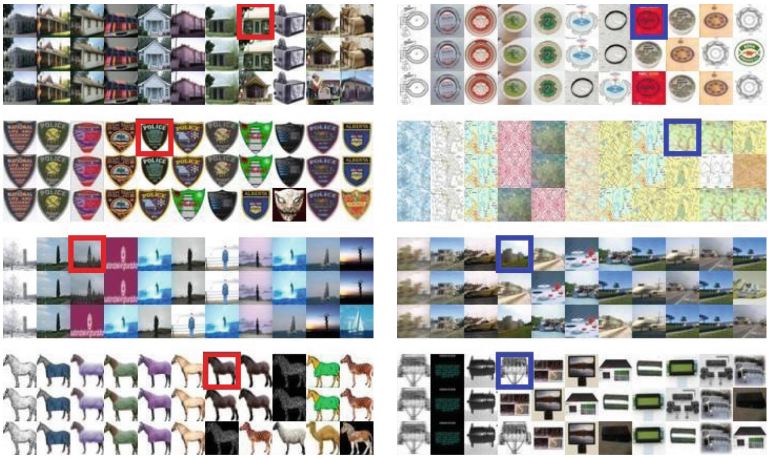
Algorithm	Volume	Time(ms)	Precision
$k = 1$			
FNNS	10000	2.640	0.978
	20000	5.407	0.992
	50000	**14.830**	**0.998**
FLANN	80000	15.809	0.939
	100000	19.447	0.954
	200000	**63.566**	**0.983**
MIH	2289901	404.3	1.000
$k = 10$			
FNNS	10000	2.603	0.968
	20000	5.337	0.988
	50000	**14.569**	**0.998**
FLANN	80000	15.647	0.802
	100000	19.583	0.841
	200000	**67.699**	**0.934**
MIH	3173628	558.5	1.000
$k = 50$			
FNNS	10000	3.262	0.947
	20000	6.308	0.979
	50000	**16.175**	**0.995**
FLANN	80000	15.908	0.632
	100000	19.939	0.696
	200000	**67.609**	**0.868**
MIH	3735299	665.3	1.000

Table 7. LSH 79 M Data

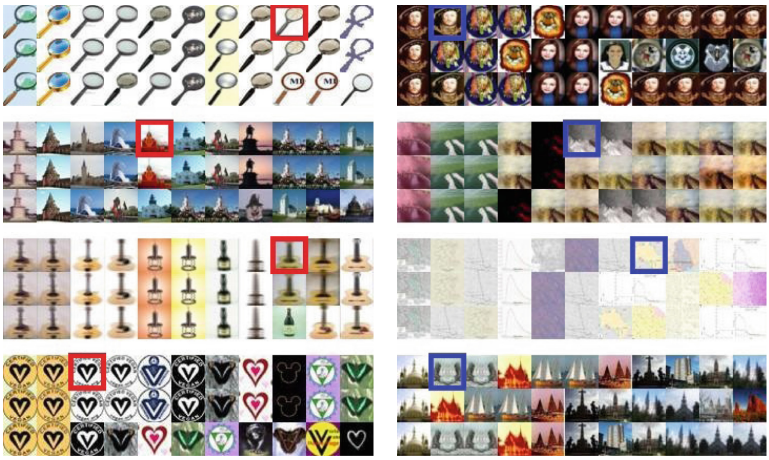
Algorithm	Volume	Time(ms)	Precision
$k = 1$			
FNNS	20000	9.682	0.774
	50000	25.215	0.862
	100000	**53.279**	**0.909**
FLANN	80000	36.267	0.734
	100000	44.545	0.756
	200000	**91.143**	**0.824**
MIH	23172087	7042.2	1.000
$k = 10$			
FNNS	20000	9.011	0.784
	50000	23.288	0.881
	100000	**49.710**	**0.930**
FLANN	80000	40.065	0.383
	100000	49.659	0.413
	200000	**92.129**	**0.527**
MIH	37399803	11513.3	1.000
$k = 50$			
FNNS	20000	11.113	0.750
	50000	28.187	0.859
	100000	**58.036**	**0.916**
FLANN	80000	40.467	0.194
	100000	47.267	0.219
	200000	**91.466**	**0.318**
MIH	41137255	12583.8	1.000

Table 8. MLH 79 M Data

Algorithm	Volume	Time(ms)	Precision
$k = 1$			
FNNS	20000	8.150	0.969
	50000	21.911	0.989
	100000	**47.077**	**0.995**
FLANN	80000	36.120	0.869
	100000	48.690	0.884
	200000	**75.446**	**0.929**
MIH	11380955	3396.6	1.000
$k = 10$			
FNNS	20000	8.227	0.961
	50000	22.026	0.987
	100000	**47.036**	**0.995**
FLANN	80000	40.873	0.600
	100000	49.348	0.637
	200000	**76.483**	**0.755**
MIH	20314128	5994.4	1.000
$k = 50$			
FNNS	20000	10.522	0.940
	50000	26.882	0.980
	100000	**55.765**	**0.992**
FLANN	80000	39.798	0.354
	100000	46.901	0.398
	200000	**73.510**	**0.554**
MIH	24028084	6697.4	1.000



(a) MLH 79M positive examples (b) MLH 79M negative examples



(c) LSH 79M positive examples (d) LSH 79M negative examples

Fig. 1. The three lines of each group represent the ground truth, the result of our approach and the result of FLANN, respectively. The leftmost image in each case is the query. For positive examples (the left-hand side), the red box indicates the sample that our approach hits, but FLANN misses. For negative examples (the right-hand side), the blue box indicates the sample that our approach misses, but FLANN hits (Color figure online).

performance to our approach when k is small while our approach, with a larger k, only takes about half search time cost to achieve a 0.99 precision.

Table 2 shows the performance on 512-bit binary codes. For a large binary code length, our approach significantly outperforms the other two methods. Compared to FLANN, our approach achieves a 0.99 precision in merely 1/3

search time cost. MIH accesses most of the data points in the search database to get the exact nearest neighbors, which makes it too time-consuming.

From Tables 3, 4, 5, 6, 7 and 8, experiments on the 80 million tiny images dataset show that our approach is scalable to large-scale search databases. Among 79M candidates, FLANN only has a 0.318 precision for LSH and a 0.554 precision for MLH when $k = 50$ while our approach can reaches a 0.916 precision for LSH and a 0.992 precision for MLH with less search time cost. However, MIH takes extremely high search time cost to retrieve the exact nearest neighbors.

Besides, Fig. 1(a) and (c) show positive examples that our approach outperforms FLANN while Fig. 1(b) and (d) show negative examples that FLANN outperforms our approach.

4.3 Analysis

Graph Construction Costs. The construction of the neighborhood graph and the bridge graph is the major part of extra computation compared to FLANN, but it is an offline task which is performed only once, thus acceptable. Exact neighborhood search for each data point in the search database may be impractical especially for large-scale search databases, therefore in experiments, we adopt FLANN as an approximate algorithm to build the neighborhood graph among the search database.

Graph Storage Costs. Both the neighborhood graph and the bridge graph are organized by attaching an adjacent list to each data point or bridge vector. By analyzing the total number of bridge vectors and adjacent lists, we can easily derive that the additional storage is $O(nd + Nk + n^m b)$, with N the number of data points, k the length of data point adjacent lists, n^m the number of bridge vectors and b the length of bridge vector adjacent lists.

Advantages over FLANN and MIH. Compared to FLANN and MIH, our approach enjoys two-fold advantages. On the one hand, both the neighborhood graph and the bridge graph structure provide an efficient way to retrieve high-quality NN candidates thanks to the close relationship built on the graph. On the other hand, candidates retrieved in such a manner are usually better than those retrieved by FLANN because the neighborhood graph provides a better order of data access. FLANN does not produce as good performance as our approach especially in large-scale search databases and the precision is very low even when a large number of data points are accessed. For MIH, the index construction and the query process are very time-consuming and therefore impractical for large-scale search databases.

5 Conclusions

In this paper, we generalize the algorithm in [15] to the Hamming space with an alternative version of k-means clustering based on the Hamming distance. The simple approach also inspires later research related to the Euclidean space and

the Hamming space. Experiments show that our algorithm outperforms state-of-the-art approaches and has significant improvement especially in the scenarios of longer binary codes or larger databases.

Acknowledgments. Weiwei Xu is partially supported by NSFC 61322204.

References

1. Torralba, A., Fergus, R., Freeman, W.T.: 80 million tiny images: a large data set for nonparametric object and scene recognition. IEEE Trans. Pattern Anal. Mach. Intell. **30**(11), 1958–1970 (2008)
2. Norouzi, M., Punjani, A., Fleet, D.J.: Fast search in hamming space with multi-index hashing. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 3108–3115. IEEE (2012)
3. Muja, M., Lowe, D.G.: Fast Matching of Binary Features. In: 9th Conference on Computer and Robot Vision, pp. 404–410. IEEE (2012)
4. Calonder, M., Lepetit, V., Strecha, C., Fua, P.: BRIEF: binary robust independent elementary features. In: Daniilidis, K., Maragos, P., Paragios, N. (eds.) ECCV 2010, Part IV. LNCS, vol. 6314, pp. 778–792. Springer, Heidelberg (2010)
5. Leutenegger, S., Chli, M., Siegwart, R.Y.: BRISK: binary robust invariant scalable keypoints. In: IEEE International Conference on Computer Vision, pp. 2548–2555. IEEE (2011)
6. Rublee, E., Rabaud, V., Konolige, K., Bradski, G.: ORB: an efficient alternative to SIFT or SURF. In: IEEE International Conference on Computer Vision, pp. 2564–2571. IEEE (2011)
7. Norouzi, M., Blei, D.M.: Minimal loss hashing for compact binary codes. In: Proceedings of the 28th International Conference on Machine Learning, pp. 353–360 (2011)
8. Charikar, M.S.: Similarity estimation techniques from rounding algorithms. In: Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing, pp. 380–388. ACM (2002)
9. Babenko, A., Lempitsky, V.: The inverted multi-index. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 3069–3076. IEEE (2012)
10. Jegou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. IEEE Trans. Pattern Anal. Mach. Intell. **33**(1), 117–128 (2011)
11. Oliva, A., Torralba, A.: Modeling the shape of the scene: a holistic representation of the spatial envelope. Int. J. Comput. Vis. **42**, 145–175 (2001)
12. Wang, J., Wang, J., Zeng, G., Tu, Z., Gan, R., Li, S.: Scalable k-NN graph construction for visual descriptors. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 1106–1113. IEEE (2012)
13. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**, 509–517 (1975)
14. Nister, D., Stewenius, H.: Scalable recognition with a vocabulary tree. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 2161–2168. IEEE (2006)
15. Wang, J., Wang, J., Zeng, G., Gan, R., Li, S., Guo, B.: Fast neighborhood graph search using cartesian concatenation. In: IEEE International Conference on Computer Vision, pp. 2128–2135. IEEE (2013)