Efficient Querying from Weighted Binary Codes

Zhenyu Weng, Yuesheng Zhu
Communication and Information Security Laboratory, Shenzhen Graduate School, Peking University
{wzytumbler, zhuys}@pku.edu.cn

Abstract

Binary codes are widely used to represent the data due to their small storage and efficient computation. However, there exists an ambiguity problem that lots of binary codes share the same Hamming distance to a query. To alleviate the ambiguity problem, weighted binary codes assign different weights to each bit of binary codes and compare the binary codes by the weighted Hamming distance. Till now, performing the querying from the weighted binary codes efficiently is still an open issue. In this paper, we propose a new method to rank the weighted binary codes and return the nearest weighted binary codes of the query efficiently. In our method, based on the multi-index hash tables, two algorithms, the table bucket finding algorithm and the table merging algorithm, are proposed to select the nearest weighted binary codes of the query in a non-exhaustive and accurate way. The proposed algorithms are justified by proving their theoretic properties. The experiments on three large-scale datasets validate both the search efficiency and the search accuracy of our method. Especially for the number of weighted binary codes up to one billion, our method shows a great improvement of more than 1000 times faster than the linear scan.

Introduction

With the explosive growth of data, binary codes are widely used to represent the data due to their small storage and efficient computation. Given a query, the nearest binary codes can be ranked and returned efficiently by computing the Hamming distance between the query and the binary codes. BRISK (Leutenegger, Chli, and Siegwart 2011), ORB (Rublee et al. 2011), and other binary image descriptors (Balntas, Tang, and Mikolajczyk 2018) are designed to represent the image data, and successfully used in various applications, including image matching, 3D reconstruction and object recognition. End-to-end feature learning methods (Li et al. 2019; Song et al. 2018) based on the neural networks extract the binary codes from the images and are widely used in image retrieval and cross retrieval. In addition to these specific image binary codes, hashing methods (Liu et al. 2014; 2017; Lin et al. 2019; Liu et al. 2019; Wang et al. 2018) are used to map different high-dimensional feature vectors into compact binary codes. Since these feature vectors may represent image data, video data, or other multimedia data, the hashing methods can be used in various multimedia retrieval applications.

However, the number of possible Hamming distance is limited and different binary codes may share the same Hamming distance to the given binary query. To alleviate this ambiguity problem and further improve the performance of binary codes, weighted binary codes are used (Fan et al. 2013; Gordo et al. 2014; Zhang et al. 2013). By assigning bitwise weights to each bit of binary codes, the distance between a pair of binary codes is calculated by weighted Hamming distance instead of Hamming distance. For example, (Huang, Wei, and Zhang 2017; Fan et al. 2013) are designed to learn the weights for the binary image descriptors to improve their discriminative power for image matching. And (Duan et al. 2015; Weng et al. 2016) are designed to learn the weights for the binary codes generated by different hashing methods to improve their search accuracy for multimedia retrieval.

Although weighted binary codes can alleviate the ambiguity problem, querying from the binary codes by weighted Hamming distance is slower than that by Hamming distance. To accelerate the querying process from the weighted binary codes, some methods (Gordo et al. 2014) use lookup tables to compute the query-independent values in advance. However, it is still an exhaustive linear scan. Some methods (Duan et al. 2015; Norouzi, Punjani, and Fleet 2014) use Hamming distance to find the neighbors that have the smallest Hamming distance to the query and rank them according to the weighted Hamming distance. This non-exhaustive way is fast but cannot return the nearest weighted binary codes of the query accurately, resulting in a degraded performance of weighted binary codes in the application.

In this paper, we propose a new method to rank the weighted binary codes and return the nearest weighted binary codes of the query in a non-exhaustive but accurate way. The diagram of our method is shown in Fig. 1. Based on the multi-index hash tables (Norouzi, Punjani, and Fleet 2014) on the binary code substrings, our method can efficiently choose the candidates in each table and merge the candidates to select the nearest weighted binary codes of the query. Theoretical analysis is provided to prove the our

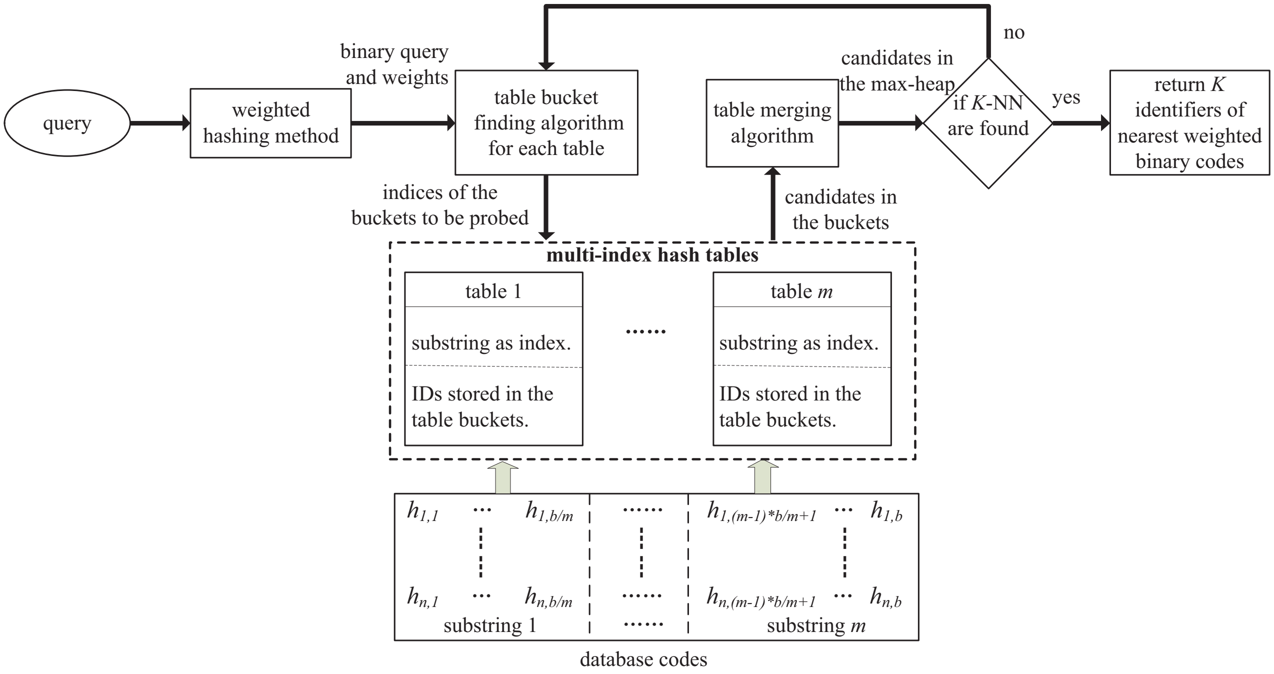


Figure 1: The diagram of our method to find the K nearest weighted binary codes of the query.

method can return the same ranking result as the linear scan does on the weighted binary codes. And the experiments show that our method is much faster than the linear scan.

Related Work

Multi-Index Hash Tables on Binary Codes

To avoid the exhaustive linear search on the binary codes, multi-index hash tables (Norouzi, Punjani, and Fleet 2014) are built to accelerate the search on the binary codes and to return the K nearest binary codes of the query in a non-exhaustive way.

In the multi-index tables (Norouzi, Punjani, and Fleet 2014), to index the binary codes from the database, m different hash tables are built based on m disjoint substrings of the binary codes as the index. If a binary code differs from the query by r bits, it is an r-neighbor of the query. And the multi-index tables can find the r-neighbors of the query efficiently by probing each table. To return the K nearest binary codes of the query, the Hamming search radius r is progressively increased to find the r-neighbors of the query, until K nearest binary codes are found.

In (Norouzi, Punjani, and Fleet 2014), the author mentioned that the multi-index tables can be used to return the top K weighted binary codes by using Hamming distance to find the candidates that have the smallest Hamming distance to the query and culling them according to the weighted Hamming distance. However, this method cannot return the K nearest weighted binary codes accurately. When increasing the search radius progressively until K neighbors are found, it guarantees that the binary codes that are found have smaller Hamming distance to the query than the ones that are not found. In contrast, it cannot guarantee that these binary codes have the smaller weighted Hamming distance than the ones that are not found. The binary codes which have the larger Hamming distance from the query may have the smaller weighted Hamming distance.

Querying from Weighted Binary Codes

As shown in Fig. 1, based on the multi-index hash tables, our method is composed of the table bucket finding algorithm and the table merging algorithm. Since we focus on finding the nearest neighbors of the query in the weighted Hamming space, in the following, we use the K-Nearest Neighbors (K-NN) of the query to denote the K nearest weighted binary codes of the query.

Table Bucket Finding Algorithm

We start with a single-index hash table and propose a table bucket finding algorithm to find the table buckets in the single-index table. To further solve the long-code problem mentioned in (Norouzi, Punjani, and Fleet 2014), we extend to the multi-index hash tables and use the table bucket finding algorithm in each table. A table merging algorithm is proposed to merge the candidates from each table.

Assume a binary query $\mathbf{q} \in \{0,1\}^b$, a binary code $\mathbf{g} \in \{0,1\}^b$ and the weight functions $w_i(\cdot)$ for each bit are given, where b is the length of the binary code and $w_i : \{0,1\} \rightarrow \mathbb{R}$. The weighted Hamming distance between the query $\mathbf{q}$ and the binary code $\mathbf{g}$ is defined as:

$$d_w(\mathbf{q}, \mathbf{g}) = \sum_{i=1}^{b} w_i(q_i \oplus g_i), \qquad (1)$$

where $\oplus$ is an xor operation, $w_i(\cdot)$ is a weight function for the i^{th} bit, q_i is the i^{th} bit of $\mathbf{q}$, and g_i is the i^{th} bit of $\mathbf{g}$.

Instead of finding the K-NN of the query $\mathbf{q}$ exhaustively, a single-index hash table is built by using the binary codes as the index of the hash table buckets. We probe the buckets in order from smallest to largest according to their weighted Hamming distance to the query, and take the identifiers as candidates in each probed table bucket until K candidates are found. These K candidates are the K-NN of the query.

The whole process of finding the buckets in order from smallest to largest can be regarded as multiple sequences combination problem (one bit represents one sequence). A algorithm (Matsui, Yamasaki, and Aizawa 2018; Babenko and Lempitsky 2015) is used to solve the multiple sequences combination problem. However, this algorithm is not suitable in this situation. The algorithm can only traverse a few sequences (e.g. 2 or 4) simultaneously to find the combination composing the bucket index that have the smallest weighted Hamming distance to the query. But in this situation, we have b sequences where b is much larger than 4 such that the traversal space is very large.

Based on the characteristic of the weighted binary codes, we propose a table bucket finding algorithm to find the bit combination which can compose the bucket index with the smallest weighted Hamming distance to the query. In the process of searching for the nearest neighbors of the query, since the query is fixed in each comparison between the query and the binary codes, the weight values for the xor result between the binary codes and the query can be pre-computed and stored. Hence, Eqn. (1) is rewritten as

$$d_w(\mathbf{g}) = \sum_{i=1}^{b} \hat{w}_i(g_i), \tag{2}$$

where g_i is the i^{th} bit of $\mathbf{g}$, $\hat{w}_i : \{0,1\} \to \mathbb{R}$ is a function to store the pre-computed weight value for the i^{th} bit and is defined as:

$$\begin{cases} \hat{w}_i(0) = w_i(0 \oplus q_i) \\ \hat{w}_i(1) = w_i(1 \oplus q_i). \end{cases} \tag{3}$$

As the input values of the function $\hat{w}_i(\cdot)$ are 0 or 1, correspondingly, there are two output values of $\hat{w}_i(\cdot)$. To construct a b-bit binary code $\mathbf{h}$= $[h_1 \ldots h_b]$ that has the smallest weighted Hamming distance (smallest sum of weights) to the query, each bit h_i of $\mathbf{h}$ is obtained as

$$h_i = \begin{cases} 0 & \hat{w}_i(0) \leq \hat{w}_i(1) \\ 1 & otherwise. \end{cases} \tag{4}$$

When the i^{th} bit of $\mathbf{h}$ is changed ($i.e.$ from 0 to 1 or from 1 to 0), we use $\bar{h}_i$ to denote the changed bit. When the bit is changed, the weight for this bit will increase. The increased weight $\Delta\hat{w}_i$ of the i^{th} bit is defined as

$$\Delta\hat{w}_i = \hat{w}_i(\bar{h}_i) - \hat{w}_i(h_i). \tag{5}$$

The bits are ranked from smallest to largest according to $\Delta\hat{w}_i$ in advance. The leftmost bit has the smallest increased weight.

After ranking the bits and constructing the smallest binary code $\mathbf{h}$, to give the buckets to be probed in order from smallest to largest, we maintain a priority queue. The top of the priority queue is the binary code that has the smallest sum of weights in the queue. $\mathbf{h}$ is the first one that is pushed into the priority queue. When taking out the top binary code $\tilde{\mathbf{h}}$ from the priority queue and probing the corresponding hash bucket, two new binary codes are constructed from $\tilde{\mathbf{h}}$ by two different operations and pushed into the priority queue, respectively.

Operation 1 is to construct a binary code by changing the unchanged bit right next to the rightmost changed bit of $\tilde{\mathbf{h}}$ if the rightmost changed bit is not at the end of the current binary code. For example, assume $\tilde{\mathbf{h}} = [h_1 \ldots \bar{h}_r \ldots h_b]$, where $\bar{h}_r$ is rightmost changed bit. Then, the new binary code is constructed as $\hat{\mathbf{h}} = [h_1 \ldots \bar{h}_r\bar{h}_{r+1} \ldots h_b]$.

Operation 2 is to construct a binary code by moving the rightmost changed bit of $\tilde{\mathbf{h}}$ to the next bit if the position of the rightmost changed bit is not at the end. For example, assume $\tilde{\mathbf{h}} = [h_1 \ldots \bar{h}_r \ldots h_b]$, where $\bar{h}_r$ is rightmost changed bit. Then, the new binary code is constructed as $\dot{\mathbf{h}} = [h_1 \ldots h_r\bar{h}_{r+1} \ldots h_b]$.

For both operations, the new binary code has a larger sum of weights than the current one since the bits are ranked from smallest to largest according to Eqn.(5) in advance. It should be noted that for the initial binary code $\mathbf{h}$ which has no changed bit, only the first operation is permitted, which means to change the first bit of the binary code.

The pseudocode for querying with the single-index hash table is shown in Alg. 1. Init() is a function that constructs the binary code $\mathbf{h}$ which has the smallest sum of weights according to the query $\mathbf{q}$ and the weights $\mathbf{w}$, and generates an order that denotes the positions of bits from smallest to largest according to Eqn.(5). Operation1() and Operation2() are two functions corresponding to above two operations to generate the new binary codes, respectively.

Algorithm 1 Querying with single-index hash table

```
Input: $\mathbf{q}$, $table$, $K$, weights $\mathbf{w}$
Output: $u$                                   ▷ a set of ranked identifiers
$u \leftarrow \emptyset$
$pri\_que \leftarrow \emptyset$                                ▷ priority queue
$[pri\_que, order] \leftarrow$ Init($\mathbf{q}$, $\mathbf{w}$)
while $|u| < K$ do
    $code \leftarrow pri\_que$.top()
    $pri\_que$.pop()              ▷ remove top item from queue
    $pri\_que$.push(Operation1($code$, $order$))
    $pri\_que$.push(Operation2($code$, $order$))
    $\hat{u} \leftarrow table$.bucket($code$)   ▷ identifiers in the bucket
    $u$.extend($\hat{u}$)
end while
```

To prove that our algorithm can always find the binary code that has the smallest sum of weights among the unprobed binary codes, we begin with the following corollary.

Corollary 1 : Every binary code can be generated by above two operations.

Proof. A detailed proof is provided in Appendix A in the supplemental material.

We prove the correctness of our algorithm as follows:

Theorem 1 : The binary code that has the smallest sum of weights among the un-probed binary codes is always in the priority queue.

Proof. A detailed proof is provided in Appendix B in the supplemental material.

Table Merging Algorithm

As described in (Norouzi, Punjani, and Fleet 2014), when the length of the binary code increases, the range of the table index expands and there are more table buckets in the table where a lot of buckets are empty. Traversing these empty table buckets is inefficient.

To solve this problem, following (Norouzi, Punjani, and Fleet 2014), m different hash tables are built based on the m disjoint substrings of the binary codes as the index in our method. The length of each substring is $\lceil b/m \rceil$ or $\lfloor b/m \rfloor$. For convenience, we assume that b can be divided by m, and that the substrings comprise continuous bits.

Before describing the following table merging algorithm, we define $f : \{0,1\}^{b/m} \to \mathbb{R}$ as a function to calculate the sum of weights of the substring $\mathbf{s}$ in the table, which is:

$$f(\mathbf{s}) = \sum_{j=1}^{b/m} \hat{w}_j(s_j), \tag{6}$$

where s_j is the j^{th} bit of $\mathbf{s}$ and $\hat{w}_j$ is the weight function for the j^{th} bit in the corresponding table.

When given a query, each table maintains a priority queue according to the sum of weights of the corresponding substring. The priority queues operate the same as in Alg. 1 to find the un-probed bucket which is indexed by the corresponding substring and has the smallest sum of weights. Then, in each round we take out the top substring of each priority queue. By treating the substring as the index of the hash table bucket in the corresponding table, we can probe the table buckets and take the identifiers in each bucket as the candidates.

To merge the candidates from each bucket and determine if the K-NN of the query are found, a K-size max-heap is built to filter the candidates. The root node of the max-heap has the largest sum of weights in the heap. Assume the node $\mathbf{r} \in \{0,1\}^b$ in the max-heap is in the form of $\mathbf{r} = [\mathbf{r}_1, \ldots, \mathbf{r}_m]$ where $\mathbf{r}_i$ is the substring of $\mathbf{r}$ in the i^{th} table. And a function $g : \{0,1\}^b \to \mathbb{R}$ to calculate the sum of weights of the node is defined as

$$g(\mathbf{r}) = \sum_{i=1}^{m} f(\mathbf{r}_i). \tag{7}$$

For each round, when the identifiers are taken from each table, they are compared to the root node in the max-heap. If an identifier $\hat{\mathbf{r}}$ has a smaller sum of weights than the root node $\mathbf{r}$ ($i.e.$ $g(\hat{\mathbf{r}}) < g(\mathbf{r})$), the root node is thrown away and the identifier is inserted into the max-heap. The process continues for multiple rounds until the root node of the max-heap is smaller or equal to a threshold.

In detail, assume there are m tables and a b-bit binary code $\mathbf{h}$ is partitioned into m disjoint substrings $\mathbf{s}$. When the top substring $\mathbf{s}_i$ of the i^{th} priority queue is taken out, the queue will have the new top substring $\hat{\mathbf{s}}_i$. The associated identifiers from the table bucket $\mathbf{s}_i$ of the i^{th} table are taken out and compared with the root node of the max-heap. The sum of weights from the top substring of each current priority queue is calculated as

$$S = \sum_{i=1}^{m} f(\hat{\mathbf{s}}_i). \tag{8}$$

If the max-heap has K nodes and the root node $\mathbf{r}$ of the max-heap is smaller or equal to the sum of weights from the top substring of each current priority queue ($i.e.$ $g(\mathbf{r}) \leq S$), the top K nearest neighbors are found and the process stops.

We prove that the table merging algorithm can find the K-NN of the query with **Theorem 2**.

Theorem 2 : The binary codes that are found and stored in the max-heap have the smallest sum of weights among all binary codes.

Proof. A detailed proof is provided in Appendix C in the supplemental material.

We can further accelerate the searching process by reducing the number of the hash table buckets to be probed. In every round, assume the priority queues are ranked in some order $\mathbf{s}_{order}$. If the top substrings from the first j queues are taken out, define the current sum of weights as below

$$\hat{S} = \sum_{i=1}^{j} f(\hat{\mathbf{s}}_{order[i]}) + \sum_{i=j+1}^{m} f(\mathbf{s}_{order[i]}). \tag{9}$$

Obviously, $\hat{S} \leq S$ according to Eqn. (7). The searching process terminates when $g(\mathbf{r}) \leq \hat{S}$. It has been proved that the binary codes that are found are the smallest among all the binary codes. From the equation, we can see that when the substring in the i^{th} queue is taken out, $\hat{S}$ will increase Δf_i, which is defined as

$$\Delta f_i = f(\hat{\mathbf{s}}_i) - f(\mathbf{s}_i). \tag{10}$$

We want to make $\hat{S}$ smallest among all the orders such that the root node of the max-heap can be smaller or equal to $\hat{S}$ faster. Hence, the order can be obtained by ranking the priority queues from smallest to largest according to Eqn.(10).

The pseudocode for querying with multi-index hash tables is shown in Alg. 2. Init(), Operation1() and Operation2() are the same functions as in the Alg. 1. m denotes the number of substrings for the binary code. $table[]$, $pri\_que[]$, $order[]$ denotes a set of tables, a set of priority queues, and a set of bit rankings for each substring, respectively. Split() is a function that splits the binary code and the weights into m parts. max_heap.satisfied() denotes whether max-heap has K nodes and the root node of the max-heap satisfies the stopping criterion. que_order denotes the order of the priority queues to be checked. Sort() is a function that determines the ranking of the priority queues from smallest to largest according to Eqn.(10). As our method performs the querying process from the weighted binary codes based on the multi-index hash tables, we call it Multi-Index Weighted Querying (MIWQ).

Algorithm 2 Querying with multi-index hash tables

```
Input: $\mathbf{q}, table[], K, m$, weights $\mathbf{w}$
Output: $max\_heap$
$max\_heap \leftarrow \emptyset$
$[pri\_que[], order[]] \leftarrow$ Split(Init($\mathbf{q}$,$\mathbf{w}$),$m$)
while !$max\_heap$.satisfied() do
    for $i \leftarrow 1$ to $m$ do
        $code[i] \leftarrow pri\_que[i]$.top()
        $pri\_que[i]$.pop()
        $pri\_que[i]$.push(Operation1($code[i], order[i]$))
        $pri\_que[i]$.push(Operation2($code[i], order[i]$))
    end for
    $que\_order \leftarrow$ Sort($pri\_que[]$.top(), $code[]$)
    for $i \leftarrow 1$ to $m$ do
        $cur = que\_order[i]$
        $max\_heap$.insert($table[cur]$.hash($code[cur]$))
        if $max\_heap$.satisfied() then
            break
        end if
    end for
end while
```

Table 1: The precision results on Places205.

bit	method	precision (%)		
		1-NN	10-NN	100-NN
32	Baseline	21.56	22.23	19.83
	MIH	22.81	23.20	20.86
	Linear Scan	25.29	24.97	22.64
	MIWQ	**25.29**	**24.97**	**22.64**
64	Baseline	31.17	29.71	26.67
	MIH	31.41	30.27	27.46
	Linear Scan	34.46	31.96	28.72
	MIWQ	**34.46**	**31.96**	**28.72**

EXPERIMENTS

Datasets and Environment

The experiments are performed on the three datasets: Places205, GIST1M and SIFT1B.

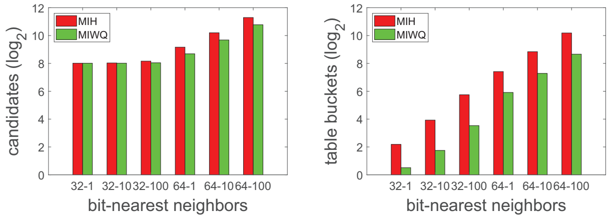
The Places205 dataset (Zhou et al. 2014) is a scene-centric dataset with 205 scene categories. For each category, we randomly choose 5,000 images for search and 50 images as queries. Hence, we have 1,025,000 images for search and 10,250 queries. Each image is represented by a 128-D feature (Cakir et al. 2017). The features are extracted from the fc7 layer of AlexNet (Krizhevsky, Sutskever, and Hinton 2012) pre-trained on ImageNet and reduced to 128 dimensions by PCA.

GIST1M dataset (Jegou, Douze, and Schmid 2011) contains 1 million 960-D GIST descriptors (Oliva and Torralba 2001) which are global descriptors, and extracted from Tiny image set (Torralba, Fergus, and Freeman 2008). The dataset contains 1000 queries.

SIFT1B dataset (Jegou, Douze, and Schmid 2011) contains 1 billion 128-D SIFT descriptors (Lowe 2004) and 10000 queries.

Table 2: The precision results on GIST1M.

bit	method	precision (%)		
		1-NN	10-NN	100-NN
32	Baseline	11.90	7.57	5.11
	MIH	12.10	8.46	5.92
	Linear Scan	13.20	9.94	7.31
	MIWQ	**13.20**	**9.94**	**7.31**
64	Baseline	21.10	14.47	10.16
	MIH	21.50	16.11	11.55
	Linear Scan	24.90	19.14	13.65
	MIWQ	**24.90**	**19.14**	**13.65**



(a) candidates (b) buckets

Figure 2: Comparison between MIWQ and MIH on SIFT1B.

In the experiments, to evaluate the efficiency and the accuracy of different querying methods on the weighted binary codes, the classical data-independent hashing algorithm Locality-Sensitive Hashing (LSH) (Andoni and Indyk 2006) is used to map high-dimensional vectors into binary codes, and a weighted hashing method, Asymmetric Distance (Asym) (Gordo et al. 2014) is used to generate the weights for each bit of binary codes. All the experiments are run on a single core Intel Core-i7 CPU with 32GB of memory. The comparison of the querying methods on other binary codes and other weights is provided in Appendix D in the supplementary material to show the generality of our method.

Comparison to Different Querying Methods

Precision@K is usually used to measure the accuracy of the approximate K-NN search. (Wang et al. 2018; Matsui, Yamasaki, and Aizawa 2018). Here, we use precision@K to evaluate whether our method can return the same results as the linear scan returns, and compare the performance of weighted binary codes in the approximate K-NN search with that of binary codes. The precision@K is defined as the fraction of the true retrieved neighbors to the retrieved neighbors. It is formulated as follows

$$precision@K = \frac{the\ true\ retrieved\ neighbors}{K} \tag{11}$$

For Places205, the ground truth refers to as the true neighbors the identifiers that have the same label as the query. For GIST1M and SIFT1B, the ground truth refers to as the true neighbors the top 1000 identifiers selected by linear scan with the Euclidean distance from the query in the original space, $i.e.$ Euclidean space.

The precision@K results on Places205 and GIST1M are shown in Table 1 and Table 2, respectively. In the ta-

Table 3: The average time for the query on Places205.

bit	method	speed-up factors for K-NN					
		1-NN		10-NN		100-NN	
		time(ms)	speed-up factor	time(ms)	speed-up factor	time(ms)	speed-up factor
32	Linear Scan	23.06	1.0	23.06	1.0	23.06	1.0
	MIH	0.11	209.6	0.21	109.8	0.52	44.3
	MIWQ	**0.11**	**209.6**	**0.21**	**109.8**	**0.55**	**41.9**
64	Linear Scan	33.83	1.0	33.83	1.0	33.83	1.0
	MIH	0.51	66.3	1.03	32.84	2.4	14.0
	MIWQ	**0.63**	**53.6**	**1.42**	**23.82**	**3.65**	**9.2**

Table 4: The average time for the query on GIST1M.

bit	method	speed-up factors for K-NN					
		1-NN		10-NN		100-NN	
		time(ms)	speed-up factor	time(ms)	speed-up factor	time(ms)	speed-up factor
32	Linear Scan	22.17	1.0	22.17	1.0	22.17	1.0
	MIH	0.1	221.7	0.16	138.5	0.37	59.9
	MIWQ	**0.12**	**184.7**	**0.25**	**88.6**	**0.77**	**28.7**
64	Linear Scan	39.36	1.0	39.36	1.0	39.36	1.0
	MIH	0.87	45.2	1.65	23.8	3.15	12.4
	MIWQ	**1.84**	**21.3**	**3.9**	**10.0**	**8.17**	**4.8**

Table 5: The average time for the query on Places205 with longer binary codes.

bit	method	speed-up factors for K-NN					
		1-NN		10-NN		100-NN	
		time(ms)	speed-up factor	time(ms)	speed-up factor	time(ms)	speed-up factor
128	Linear Scan	65.93	1.0	65.93	1.0	65.93	1.0
	MIH	2.08	31.6	4.19	15.7	8.50	7.7
	MIWQ	**3.13**	**21.0**	**6.71**	**9.8**	**15.25**	**4.3**
256	Linear Scan	103.92	1.0	103.92	1.0	103.92	1.0
	MIH	5.68	18.2	11.80	8.8	21.22	4.8
	MIWQ	**9.30**	**11.1**	**21.97**	**4.7**	**39.28**	**2.6**

Table 6: The precision results for the query on SIFT1B.

bit	method	precision (%)		
		1-NN	10-NN	100-NN
32	Baseline	1.61	1.66	1.77
	MIH	3.04	2.04	1.48
	Linear Scan	3.25	2.77	2.66
	MIWQ	**3.25**	**2.77**	**2.66**
64	Baseline	7.62	8.26	9.07
	MIH	25.18	20.22	10.52
	Linear Scan	26.77	21.18	14.13
	MIWQ	**26.77**	**21.18**	**14.13**

bles, Baseline denotes querying from the binary codes according to Hamming distance, Linear Scan denotes querying from the binary code by the linear scan according to weighted Hamming distance, and Multi-Index Hashing (MIH) (Norouzi, Punjani, and Fleet 2014) is a non-exhaustive but inexact querying method for the binary codes according to weighted Hamming distance. These querying methods are all implemented in C++. For MIH and our method, MIWQ, we use the same heuristic (Norouzi, Punjani, and Fleet 2014) to determine the number of the substrings m, which is $b/log_2 n$ where b is the length of the binary code and n is the data size. According to the results, MIH can achieve higher search accuracy than Baseline, but is inferior to MIWQ. As MIH cannot return the K nearest weighted binary codes accurately, MIH is inferior to MIWQ. Since MIWQ achieves the same search accuracy as Linear Scan, it shows that MIWQ can return the K nearest weighted binary codes of the query accurately.

The speed-up factor is used to measure how fast our method and MIH are compared to the linear scan on the weighted binary codes. The speed-up factor is defined as dividing the run-time cost of the linear scan by the run-time cost of the test method, which is formulated as follows

$$speed{-}up\ factor = \frac{time\ cost\ of\ linear\ scan}{time\ cost\ of\ test\ method} \qquad (12)$$

Table 3 and Table 4 shows the average time for each query of returning the different amounts of Nearest Neighbors (NN) on Places205 and GIST1M, respectively. Linear

Table 7: The average time for the query on SIFT1B.

bit	method	speed-up factors for K-NN					
		1-NN		10-NN		100-NN	
		time(ms)	speed-up factor	time(ms)	speed-up factor	time(ms)	speed-up factor
32	Linear Scan	22380.15	1.0	22380.15	1.0	22380.15	1.0
	MIH	6.13	3650.9	6.14	3644.9	6.38	3507.8
	MIWQ	**6.15**	**3639.0**	**6.31**	**3546.7**	**6.53**	**3427.3**
64	Linear Scan	43623.33	1.0	43623.33	1.0	43623.33	1.0
	MIH	7.77	5614.3	11.42	3819.9	22.96	1899.9
	MIWQ	**7.78**	**5607.1**	**12.65**	**3448.4**	**32.17**	**1356.0**

Table 8: Time cost (ms) on SIFT1B.

bit-K NN	MIWQ	PQTable	$\mathrm{PQTable}_{\mathrm{max\_heap}}$
32-1	6.15	51.31	50.44
32-10	6.31	51.22	50.48
32-100	6.53	51.43	50.55
64-1	7.78	57.26	54.53
64-10	12.65	64.55	61.21
64-100	32.17	92.11	85.33

Scan is accelerated by adopting the look-up tables (Gordo et al. 2014). From the results, we can see that MIH and MIWQ are both faster than Linear Scan in all the cases. MIWQ is comparable or a litter inferior to MIH in the average time. From the tables, we can see that the query time of both MIH and MIWQ for 100-NN is larger than that for 1-NN. To return more neighbors about the query, more buckets need to be probed, resulting in a larger time cost.

Case of Longer Binary Codes

In the above experiments, we analyze the performance of our method for 32 bits and 64 bits, which are the commonly used length of binary codes for the hashing methods. In some situations, longer binary codes (such as 128 bits and 256 bits) are used to achieve higher search accuracy but with additional storage cost. Here, we analyze the performance of our method in the case of long binary codes.

Table 5 shows the average time for the query on Places205. For 128 bits and 256 bits, MIWQ can still accelerate the search on the binary codes. By comparing Table 5 to Table 3, the speed-up factors for 128 bit and 256 bits are smaller than the ones for 32 bits and 64 bits.

Case of Larger Dataset

Table 6 shows the precision@K results on SIFT1B. From the results, we can see that MIWQ still achieves better search accuracy than MIH and Baseline.

Table 7 shows the average time for the query on SIFT1B. From the results, we can see that MIH and MIWQ both have a large improvement on the speed compared to linear scan. MIWQ achieves almost the same time cost as MIH does for 32 bits and 64 bits. In the aspect of comparing candidates, MIWQ compares candidates by using weighted Hamming distance, while MIH compares candidates by using Hamming distance at first and then culls candidates by using weighted Hamming distance. Hence, MIH has a smaller time cost than our method. However, the factors to affect the search efficiency is not only the distance computation, but also the number of the candidates to be compared and the number of the table buckets to be probed. The number of candidates and the number of table buckets are shown in Fig. 2. Since the average number of candidates and table buckets in our method are both smaller than those in MIH, our method can be almost as fast as MIH.

PQTable

Recently, PQTable (Matsui, Yamasaki, and Aizawa 2018) is proposed to perform an efficient search for Product Quantization (PQ) (Jegou, Douze, and Schmid 2011) which is another encoding method. With some modifications, PQTable can be used for querying from the weighted binary codes. The binary codes are split into disjoint parts each of which consists of continuous 8-bit binary codes. Then, each part can be regarded as a codebook, and PQTable is applied.

Since PQTable and our method both can return the nearest weighted binary codes of the query accurately, we compare them with respect to the running time. As PQTable is also based on the multi-index tables, to further explore the difference between our method and PQTable, our table merge algorithm is applied to PQTable and replaces the table merge algorithm of PQTable, which is dubbed as $\mathrm{PQTable}_{\mathrm{max\_heap}}$. Table 8 shows the time comparison between MIWQ, PQTable and $\mathrm{PQTable}_{\mathrm{max\_heap}}$. According to the results, our table merging algorithm is faster than that of PQTable, which shows that our table merging algorithm can terminate the process by determining whether the K-NN of the query have been found faster than that of PQTable. Comparing MIWQ with $\mathrm{PQTable}_{\mathrm{max\_heap}}$, with same table merging algorithm, MIWQ is faster than $\mathrm{PQTable}_{\mathrm{max\_heap}}$, especially for the 32-bit case. As MIWQ and $\mathrm{PQTable}_{\mathrm{max\_heap}}$ both perform the exact K-NN search, the order of table buckets to be probed is the same. The difference between them is the process to find the next hash table bucket. Since our method exploits the characteristics of the weighted binary codes, the bucket candidate space to traverse from our method is smaller than that of PQTable. Hence, our method can find the next smallest un-probed binary bucket faster than PQTable.

Conclusion

In this paper, a new querying method is proposed to return the nearest weighted binary codes of the query in a non-exhaustive way. The method consists of two algorithms, the table bucket finding algorithm and the table merging algorithm. The former one is designed to consecutively find the un-probed table buckets, and the latter one is developed to merge the candidates from each table. The experiments show that our method can produce the same querying results as linear scan does with a large time speed-up on the large-scale dataset which includes up to 1 billion data points.

Acknowledgements

This work is supported by Shenzhen Municipal Development and Reform Commission (Disciplinary Development Program for Data Science and Intelligent Computing), Key-Area Research and Development Program of Guangdong Province (#2019B010137001), NSFC-Shenzhen Robot Jointed Founding (U1613215).

References

Andoni, A., and Indyk, P. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *47th Annual IEEE Symposium on Foundations of Computer Science*, 459–468.

Babenko, A., and Lempitsky, V. 2015. The inverted multi-index. *IEEE TPAMI* 37(6):1247–1260.

Balntas, V.; Tang, L.; and Mikolajczyk, K. 2018. Binary online learned descriptors. *IEEE TPAMI* 40(3):555–567.

Cakir, F.; He, K.; Bargal, S. A.; and Sclaroff, S. 2017. Mihash: Online hashing with mutual information. In *Proceedings of the ICCV*, 437–445.

Duan, L. Y.; Lin, J.; Wang, Z.; Huang, T.; and Gao, W. 2015. Weighted component hashing of binary aggregated descriptors for fast visual search. *IEEE TMM* 17(6):828–842.

Fan, B.; Kong, Q.; Yuan, X.; Wang, Z.; and Pan, C. 2013. Learning weighted hamming distance for binary descriptors. In *Proceedings of the ICASSP*, 2395–2399.

Gordo, A.; Perronnin, F.; Gong, Y.; and Lazebnik, S. 2014. Asymmetric distances for binary embeddings. *IEEE TPAMI* 36(1):33–47.

Huang, Z.; Wei, Z.; and Zhang, G. 2017. Rwbd: Learning robust weighted binary descriptor for image matching. *IEEE TCSVT* PP(99):1–1.

Jegou, H.; Douze, M.; and Schmid, C. 2011. Product quantization for nearest neighbor search. *IEEE TPAMI* 33(1):117–128.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Proceedings of the NeurIPS*, 1097–1105.

Leutenegger, S.; Chli, M.; and Siegwart, R. Y. 2011. Brisk: Binary robust invariant scalable keypoints. In *Proceedings of the ICCV*, 2548–2555.

Li, C.; Deng, C.; Wang, L.; Xie, D.; and Liu, X. 2019. Coupled cyclegan: Unsupervised hashing network for cross-modal retrieval. In *Proceedings of the AAAI*, 176–183.

Lin, M.; Ji, R.; Liu, H.; Sun, X.; Wu, Y.; and Wu, Y. 2019. Towards optimal discrete online hashing with balanced similarity. In *Proceedings of the AAAI*, 8722–8729.

Liu, W.; Mu, C.; Kumar, S.; and Chang, S.-F. 2014. Discrete graph hashing. In *Proceedings of the NeurIPS*, 3419–3427.

Liu, X.; Deng, C.; Mu, Y.; and Li, Z. 2017. Boosting complementary hash tables for fast nearest neighbor search. In *Proceedings of the AAAI*, 4183–4189.

Liu, H.; Ji, R.; Wang, J.; and Shen, C. 2019. Ordinal constraint binary coding for approximate nearest neighbor search. *IEEE TPAMI* 41(4):941–955.

Lowe, D. G. 2004. Distinctive image features from scale-invariant keypoints. *IJCV* 60(2):91–110.

Matsui, Y.; Yamasaki, T.; and Aizawa, K. 2018. Pqtable: Nonexhaustive fast search for product-quantized codes using hash tables. *IEEE TMM* 20(7):1809–1822.

Norouzi, M.; Punjani, A.; and Fleet, D. J. 2014. Fast exact search in hamming space with multi-index hashing. *IEEE TPAMI* 36(6):1107–1119.

Oliva, A., and Torralba, A. 2001. Modeling the shape of the scene: A holistic representation of the spatial envelope. *IJCV* 42(3):145–175.

Rublee, E.; Rabaud, V.; Konolige, K.; and Bradski, G. 2011. Orb: An efficient alternative to sift or surf. In *Proceedings of the ICCV*, 2564–2571.

Song, J.; He, T.; Gao, L.; Xu, X.; Hanjalic, A.; and Shen, H. T. 2018. Binary generative adversarial networks for image retrieval. In *Proceedings of the AAAI*, 394–401.

Torralba, A.; Fergus, R.; and Freeman, W. T. 2008. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE TPAMI* 30(11):1958–1970.

Wang, J.; Zhang, T.; Song, J.; Sebe, N.; and Shen, H. T. 2018. A survey on learning to hash. *IEEE TPAMI* 40(4):769–790.

Weng, Z.; Yao, W.; Sun, Z.; and Zhu, Y. 2016. Asymmetric distance for spherical hashing. In *Proceedings of the ICIP*, 206–210.

Zhang, L.; Zhang, Y.; Tang, J.; Lu, K.; and Tian, Q. 2013. Binary code ranking with weighted hamming distance. In *Proceedings of the CVPR*, 1586–1593.

Zhou, B.; Lapedriza, A.; Xiao, J.; Torralba, A.; and Oliva, A. 2014. Learning deep features for scene recognition using places database. In *Proceedings of the NeurIPS*, 487–495.