A MULTI-BLOCK N-ARY TRIE STRUCTURE FOR EXACT R-NEIGHBOUR SEARCH IN HAMMING SPACE

Yicheng Huang[1], *Ling-Yu Duan*[*,1], *Zhe Wang*[1], *Jie Lin*[2], *Vijay Chandrasekhar*[2,3], *Tiejun Huang*[1]

[1]Institute of Digital Media, School of EECS, Peking University, China
[2]Institute for Infocomm Research, Singapore
[3]Nanyang Technological University, Singapore

ABSTRACT

This paper proposes a novel algorithm to solve the exact r-neighbour search problem in Hamming space. Existing r-neighbour search methods typically adopt hash table to index binary codes. Given a query, existing approaches search the nearest neighbours by checking all buckets of a Hamming ball centered at the query. The problem is these methods spend most of search time visiting empty buckets (lookup misses). In this paper, we adopt trie structure to index binary codes. We consider several continuous bits of a binary string as a block and use it as an atomic indexing element in trie structure, which is efficient in access speed and memory usage. Our method searches the nearest neighbours of a query by utilizing the records of nodes in trie structure to avoid lookup misses. We name the proposed indexing structure as Multi-Block N-ary Trie (MBNT). A theoretical analysis is given to prove that MBNT has less time cost than other hash-based methods. Extensive results show that MBNT outperforms state-of-the-art algorithms on several large scale benchmarks.

Index Terms— Binary Code, Trie, Index, R-Neighbour

1. INTRODUCTION

Binary representation [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] for high dimensional features has attracted a lot of attention in recent years. The goal of binary coding is to compress features into compact binary strings which could be beneficial for handling large datasets [16, 5, 2]. Although Hamming distance matching between binary codes is very fast (more than millions of matching in one second) [8, 17], when the size of dataset is extremely huge, exhaustive searching over the whole dataset is still too slow for real-time retrieval. Therefore, designing efficient indexing algorithms to improve the retrieval speed of binary string is still quite necessary for fast retrieval over large-scale datasets [18].

One way to index binary codes for nearest neighbour search is using hash tables [19, 20, 3, 6], where binary codes are directly used as indices (addresses) into hash buckets. Extension evaluations [2, 3, 6] show that this hash approach could yield a dramatic increase in search speed compared to an exhaustive linear scan. However, in practice, using hash tables is not necessarily efficient in memory usage [6], which is essentially trading time for space. Ideally, one needs to set up a hash table with 2^d buckets for indexing d-bits binary codes. When d increases to 64, the memory cost for $2^{64} \approx 10^{19}$ buckets is not affordable.

To handle long codes, Norouzi et.al proposed Multi-Index Hashing (MIH) [18] to use a group of hash tables to index the substrings of binary codes. Specifically, MIH divides binary code into multiple disjoint substrings, and then uses a hash table for each substring instead of the whole binary code. This partition strategy enables efficient indexing long codes. Experimental results show that MIH provides very promising speedup for retrieval over long-length codes, say 64, 128, or 256 bits.

One problem of hash-based methods is that, one needs to check all buckets in a Hamming ball around the query to find nearest neighbours. Given code length d and search radius r, the total number of buckets that need to examine is

$$L(d,r) = \sum_{k=0}^{r} \binom{d}{k}. \tag{1}$$

where $L(d,r)$ grows exponentially with r. When r is large, the search range increases dramatically. In practice, we notice that most buckets in hash table are empty. Visiting empty buckets, which we call lookup misses, is not necessary and wastes a lot of time.

In this paper, we propose a novel data structure, Multi-Block N-ary Trie (MBNT), to address the issues of hash-based approaches for exact r-neighbour search in Hamming space. Compared to other methods, we utilize trie structure to index binary codes to avoid lookup misses. We summarize the main contributions of this work as follows:

- We propose to index binary codes with trie structure for r-neighbour search in Hamming space. We consider continuous bits of binary codes, which we call a block, as an atomic element to generate the trie structure, which is efficient in both access speed and memory cost.
- Given a query, we search the nearest neighbours by traversing the nodes in the trie. As we only check the elements which exist in the nodes of trie structure, our method can avoid lookup misses. Theoretical analysis shows that our method has a lower time cost than hash-based methods.
- We evaluate our method on several benchmarks. Experimental results show that the proposed MBNT outperforms recent state-of-the-art methods. Specifically, MBNT is 2-4 times faster than Multi-Index Hashing (MIH), and is hundreds of times faster than exhaustive linear scan in most of the cases.

2. RELATED WORKS

There are two related nearest neighbour search problem in Hamming space, K-th nearest neighbour search [21] and r-neighbour search [22, 23]. The first is to find the K codes in dataset that are closest in Hamming space to a given query. The second problem

∗Ling-Yu Duan is the corresponding author.

is to find all codes in the dataset that are within a fixed Hamming distance of a query. Actually, these two problems are interconvertible [18]. In this paper, we focus on the latter one, i.e., r-neighbour search problem, where the formal definition is presented as follows,

Definition 1 *Given a dataset $\mathcal{B} = \{\mathbf{b}_i\}_{i=1}^n$ and a query q where $\mathbf{b_i}, q \in \{0,1\}^d$, the r-neighbours $\mathcal{D}_r(q,\mathcal{B})$ is defined as all codes in $\mathcal{B}$ which differ from q in r or less bits:*

$$\mathcal{D}_r(q,\mathcal{B}) = \{\mathbf{b_i} \in \mathcal{B} : H(\mathbf{b_i}, q) \leq r\}.$$

Here $H(.)$ denotes the Hamming distance.

In hash-based methods [19, 20, 3, 6], binary codes are directly used as addresses in hash table. Given a query q and search radius r, one needs to enumerate all r-neighbours of q and then check corresponding hash buckets to see if they exist. This method needs a big hash table with 2^d buckets to index d-bits binary codes. When d becomes large, such as 64 bits or more, the memory requirement is become infeasible [18].

To address this issue, Multi-Index Hashing (MIH) [18] partitions every binary code into m disjoint substrings, where each substring has a length of $s = d/m$ bits. According to the pigeonhole principle, if two binary codes p and q differ by r bits or less, at least one of their m substring must differ by at most $r' = \lfloor r/m \rfloor$ bits. Therefore, MIH builds a hash table T_j for each j-th substring. Give a query q with substrings $q_1, \ldots, q_m$, MIH firstly searches the r'-neighbours of q_j from T_j, denoted as $\mathcal{D}_{r'}(q_j)$. Then, combines the sets of all r'-neighbours as a candidate set $\mathcal{S} = \bigcup_j \mathcal{D}_{r'}(q_j)$, and finally, test all candidates in $\mathcal{S}$ and remain r-neighbours of q.

Consider the search cost of MIH. We first calculate the time cost for each substring, and then multiply it by m to get total result. Assume binary codes are uniformly distributed over the Hamming space. For each substring, the number of buckets need to lookup is

$$lookup_{\text{MIH}}(s) = L(s, r'). \tag{2}$$

Let n denote the number of binary codes in the database. As there are 2^s buckets in the hash table, the average entries that each bucket has should be $n/2^s$. The total number of candidates that need to be tested is

$$ctest_{\text{MIH}}(s) = \frac{n}{2^s} lookup_{\text{MIH}}(s). \tag{3}$$

Then, the total time cost for searching r-neighbours of a query q is

$$\begin{aligned} cost_{\text{MIH}}(s) &= m(lookup_{\text{MIH}}(s) + ctest_{\text{MIH}}(s)) && (4)\\ &= \frac{d}{s}(1 + \frac{n}{2^s}) lookup_{\text{MIH}}(s) && (5)\\ &= \frac{d}{s}(1 + \frac{n}{2^s}) L(s, r'). && (6) \end{aligned}$$

In most scenarios we have $n \ll 2^s$, which means the item $lookup$ dominates the cost function. On the other hand, there are many empty buckets in hash table since n typically is much less that 2^s. Checking empty buckets is not necessary and waste time. This observation motivates us to adopt trie structure to avoid lookup misses to reduce the cost at the lookup stage, i.e., reduce the value of $lookup_{\text{MIH}}$.

We notice that there are also several papers [24, 25, 26, 27, 28] for approximate nearest neighbour (ANN) search in Hamming space. But in this paper we focus on the exact r-neighbour search problem which means we must find all nearest neighbours of a query within Hamming distance r.

Algorithm 1 r-neighbour Search in Hamming Space.

Given query substrings $\{q_j\}_{j=1}^m$, MBNT $\{T_j\}_{j=1}^m$, query radius r and block parameter c, set $r' = \lfloor r/m \rfloor$. Initiate $D \leftarrow \emptyset$.
for $i \leftarrow 1$ to m **do**
  $D = D \cup$ QueryMBNT($root(T_i), 0, q_i, 0$)
Test and remove all non r-neighbour elements in D.
return D

function QUERYMBNT(u, dep, q, h)
  if $h > r'$ **then return** $\emptyset$
  if u is leaf node **then**
    return lookup $(r' - h)$-neighbours from container
  else
    $ans \leftarrow \emptyset$
    for all v is child of u **do**
      $h' \leftarrow$ Hamming distance($q[dep...dep + c - 1], v$)
      $ans = ans \cup$ QueryMBNT($v, dep + c, q, h + h'$)
    return ans

3. MULTI-BLOCK N-ARY TRIE STRUCTURE

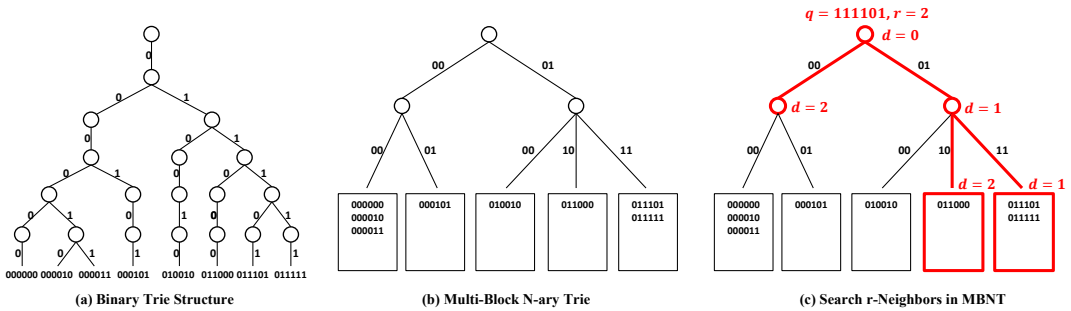
In this section, we present our method, Multi-Block N-ary Trie (MBNT) structure, to address the r-neighbour search problem in Hamming space. Our method uses the partition strategy proposed in MIH [18] to handle long-length binary strings. We divide each binary string in dataset into m disjoint substrings and index each of them individually. Next, we focus on how to design efficient indexing structure for each substring.

3.1. Indexing Binary Strings with Trie

A trie is a tree data structure where all the descendants of a node have a common prefix of the string associated with that node, and the root of the trie is associated with the empty string. For the node at depth l of a trie, it represents the set of all strings that begin with the same first l characters, and its branches are defined based on the $l+1$ character of strings. A special case of a trie for binary strings is a binary tree since the branches at each node are either 0 or 1. Figure 1(a) shows an example of a binary trie.

The MBNT structure considers c continuous bits, denoted block, as an atomic element for indexing. More precisely, given a binary string with s bits, each continuous c bits is treated as a block where it totally has $\frac{s}{c}$ blocks and each block can represent 2^c symbols. As a special case, if we set $c = 1$, the MBNT structure is become a binary trie. The reason why we combine multiple bits into a block for indexing is to make the access speed and memory usage efficient in the trie structure, since a binary trie for s-bits strings has s layers, which could be very deep when the string length is large.

After that, for each leaf node of the trie, we build a container which contains all binary strings in dataset that have common prefix corresponding to the leaf node. In practice, we only utilize the first b bits ($b \leq s$) for indexing in a trie to reduce the memory usage. Both c and b impact the performance of our proposed algorithm. We will discuss the effect of b and c in the following sections. Figure 1(b) shows an example of the proposed Multi-Block N-ary Trie where dataset $\mathcal{B} = \{000000, 000010, 000011, 000101, 010010, 011000, 011101, 011111\}$. In this example, we set $b = 4$ and $c = 2$.



(a) Binary Trie Structure (b) Multi-Block N-ary Trie (c) Search r-Neighbors in MBNT

Fig. 1. Suppose we have eight 6-bits binary strings $\mathcal{B} = \{000000, 000010, 000011, 000101, 010010, 011000, 011101, 011111\}$. (a) Example of indexing binary strings with binary trie structure. (b) Example of our proposed Multi-Block N-ary Trie structure. (c) Example of searching r-neighours in MBNT where query $q = 111101$ and search radius $r = 2$; The traversal paths in MBNT are colored in red. This figure is best viewed in color.

3.2. Searching the r-Neighbours with MBNT

Given query q and search radius r, we search the r-neighbours of q by traversing the nodes in MBNT. The traversal starts from root node with initial Hamming distance $h = 0$. When visiting one node at the l-th layer, we calculate the Hamming distance between the binary string associated with the node's path and the first l query blocks. If the path's binary string has more than r bits difference from the query, the traversal is discarded. On reaching a leaf node, we check the candidates in the container and return binary strings with Hamming distance $h \leq r$. Figure 1(c) shows an example of r-Neighbours search in MBNT. Algorithm 1 shows the pseudo-code.

3.3. Theoretical Analysis

Given size b, the Multi-Block N-ary Trie (MBNT) mostly has $p = 2^b$ leafs. Assume binary strings follow uniform distribution, we totally insert n items into trie. The probability that one leaf doesn't exist in MBNT should be [1]

$$(\frac{p-1}{p})^n \approx e^{-n/p}. \tag{7}$$

Therefore, the expected density of leaf equals to

$$density(b) = 1 - e^{-\frac{n}{2^b}}. \tag{8}$$

Let $lookup_{\text{MIH}}(s)$ denote the total lookup number of the exhausted enumeration for r-neighours of query. The expected number of lookup in MBNT is

$$lookup_{\text{MBNT}}(b,s) = density(b) * lookup_{\text{MIH}}(s) \tag{9}$$

$$= (1 - e^{-\frac{n}{2^b}}) * lookup_{\text{MIH}}(s) \tag{10}$$

where $1/density(b)$ denotes the speedup rate of our proposed MBNT for lookup stage compared with hash-based method. When $s = 32, b = 30, n = 5 * 10^7$, we have $density(b) \approx 0.045$. That is to say, 95.5% lookup items would be discarded by our MBNT algorithm.

3.4. Implement Details

In practice, we adopt a static structure, full 2^c-ary tree, to implement the proposed Multi-Block N-ary Trie (MBNT). A 1-bit boolean flag is used to indicate whether each node truly exists. The container is implemented by a static array. Parameter b is chosen to be multiples of c. Large b could be helpful to further reduce the density of leaf node but also increases the access time in trie. In addition, we noticed that setting c more than 4 is not beneficial for performance. In this work, we set $b = 30, c = 3$ or $b = 28, c = 4$ for 32-bit substrings.

[1] $(1 - 1/p)^p \approx 1/e$ since p is quite large.

4. EXPERIMENT

We run our experiments on Intel Xeon CPU with 2560KB L2 cache and 128GM RAM. The MBNT is implemented in C++.

4.1. Datasets and Baselines

We evaluate our method on three large scale benchmarks SIFT-50M, SIFT-200M and GIST-79M, where SIFT-50M and SIFT-200M are subsets of BIGANN SIFT dataset [29], and GIST-79M is a dataset of 79M GIST feature extracted from 80 million tiny images [30]. We utilize LSH [24] to binarize features into binary codes of different lengths 64 bits and 128 bits. For each dataset, we randomly select 100 elements as queries.

We compare our method with the state-of-the-art Multi-Index Hashing [18] (MIH) and exhaustive linear scan. The implementations of MIH and linear scan are from publicly available source codes with recommended parameter settings [18]. We set $b = 30, c = 3$ and $s = 32$ in MBNT for evaluation.

4.2. Query Time

Figure 2 shows the query time of r-neighbour search on dataset SIFT-50M, GIST-79M and SIFT-200M. Features are compressed into 64 bits on SIFT-50M and GIST-79M, and 128 bits on SIFT-200M. MBNT consistently outperforms MIH and linear scan. Compared with MIH, MBNT achieves more than 2x speedup in most cases. For dataset SIFT-50M, when search radius $r \geq 12$, the speedup of MBNT is up to 4x than MIH. Similar trends are observed on GIST-79M and SIFT-200M. Both MBNT and MIH outperform linear scan. Compared to linear scan, MBNT achieves significant improvement in retrieval speed especially for small search radius. When r is small e.g. $r \leq 10$, MBNT can be hundreds of time faster than linear scan.

4.3. Time Cost Analysis

Figure 3 shows several important values of MBNT and MIH. In this figure, as our method mainly focus on improving speed on lookup,

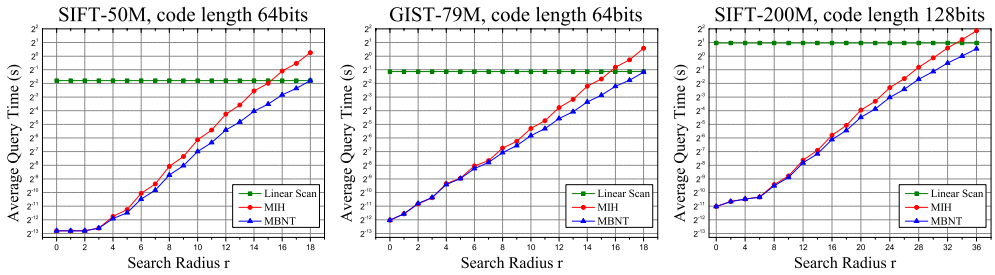


Fig. 2. Comparison of query time for different methods on SIFT-50M, GIST-79M and SIFT-200M datasets. The vertical coordinates are in log base 2.

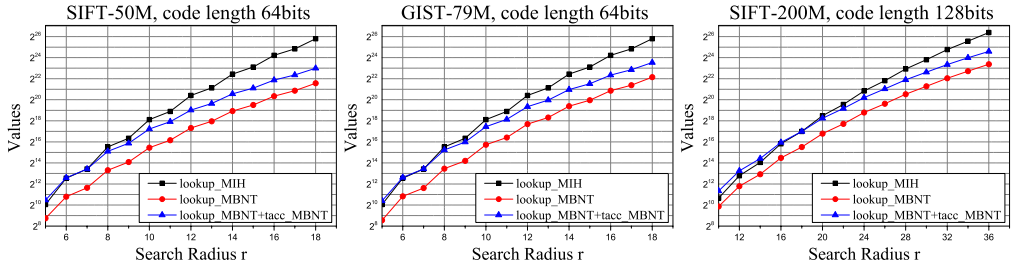


Fig. 3. Important statistic results for MBNT and MIH on SIFT-50M, GIST-79M and SIFT-200M datasets. The vertical coordinates are in log base 2.

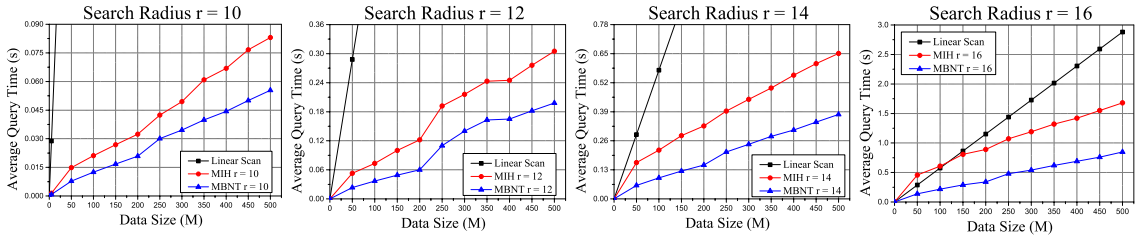


Fig. 4. Comparison of query time for different methods at different dataset sizes. We use dataset SIFT-1B from BIGANN [29] for evaluation. The length of binary codes is 64 bits.

we only consider the results on lookup stage where $lookup_{\text{MBNT}}$ and $lookup_{\text{MIH}}$ denote the time cost of MBNT and MIH, respectively. First, we can see that $lookup_{\text{MBNT}}$ is always less than $lookup_{\text{MIH}}$, which shows that MBNT efficiently reduces lookup misses of hash-based methods. As MBNT utilizes trie structure for indexing, it needs extra time to access the trie nodes during search, which is denoted bt $tacc_{\text{MBNT}}$. Therefore, $lookup_{\text{MBNT}} + tacc_{\text{MBNT}}$ indicates the actual time cost at lookup stage for MBNT. Since $lookup_{\text{MBNT}} + tacc_{\text{MBNT}} \leq lookup_{\text{MIH}}$ in most of cases, it demonstrates that MBNT has improved the speed of lookup over MIH. Moreover, as search radius r increases, the speedup rate of MBNT at lookup stage becomes more significant. The reason is that when r becomes large the number of empty buckets checked by MIH also grows up. By avoiding lookup misses, MBNT can efficiently reduce time cost in lookup stage and improve retrieval speed. The experimental results are consistent with our previous analysis.

4.4. Effect of Dataset Size

Figure 4 shows the retrieval speed of state-of-the-art methods at different dataset size from 50 million to 500 million. MBNT consistently outperforms MIH and linear scan at all dataset sizes. We notice that when dataset size is small, say 50 million or 100 million, MIH is slower than linear scan occasionally. As dataset size increases, the gap between MBNT and MIH becomes larger. For example, at dataset size 50 million with search radius $r = 14$, the average query time of MBNT is about 0.1 second less than MIH, it becomes 0.25 second when dataset size is 500 million. This demonstrates that MBNT handles very large scale dataset better than other methods.

5. CONCLUSION

MBNT is more efficient than hash-based methods for indexing binary codes for r-neighbour search. By utilizing the records of nodes in trie, MBNT avoids lookup misses and reduces time cost in the lookup stage. We provide a group of quantitative results to verify the effectiveness of the proposed MBNT. Experimental results are consistent with our theoretical analysis. Extensive evaluations show that MBNT significantly outperforms state-of-the-art methods on several widely used benchmarks. What's more, it's demonstrated that MBNT handles very large scale dataset better than other methods.

6. ACKNOWLEDGMENTS

This work was partially supported by grants from National NaturalScience Foundation of China (U1611461, 61661146005) and National Hightech R&D Program of China (2015AA016302).

7. REFERENCES

[1] W. Liu, J. Wang, S. Kumar, and S.-F. Chang, "Hashing with graphs," *ICML*, 2010.

[2] Yair Weiss, Antonio Torralba, and Rob Fergus, "Spectral hashing," *In NIPS*, 2008.

[3] Brian Kulis and Trevor Darrell, "Learning to hash with binary reconstructive embeddings," *NIPS*, 2009.

[4] Jun Wang, Sanjiv Kumar, and Shih-Fu Chang, "Semi-supervised hashing for scalable image retrieval," *CVPR*, 2010.

[5] Yunchao Gong and Svetlana Lazebnik, "Iterative quantization: A procrustean approach to learning binary codes," *CVPR*, 2011.

[6] Mohammad Norouzi and David Fleet, "Minimal loss hashing for compact binary codes," *In ICML*, 2011.

[7] Weihao Kong and Wu-Jun Li, "Isotropic hashing," *In NIPS*, 2012.

[8] Kaiming He, Fang Wen, and Jian Sun, "K-means hashing: an affinity-preserving quantization method for learning binary compact codes," *CVPR*, 2013.

[9] Wei Liu, Cun Mu, Sanjiv Kumar, and Shih-Fu Chang, "Discrete graph hashing," *In NIPS*, 2014.

[10] Yan Xia, Kaiming He, Pushmeet Kohli, and Jian Sun, "Sparse projections for high-dimensional binary codes," *CVPR*, 2015.

[11] L. Jie, O. Morere, V. Chandrasekhar, A. Veillard, and H. Goh, "Co-sparsity regularized deep hashing for image instance retrieval," *ICIP*, 2016.

[12] J. Lin, O. Morere, V. Chandrasekhar, A. Veillard, and H. Goh, "Deep hash: Getting depth, reguarlization and fine tuning right," *Arxiv*, 2015.

[13] Zhe Wang, Ling-Yu Duan, Jie Lin, Xiaofang Wang, Tiejun Huang, and Wen Gao, "Hamming compatible quantization for hashing," *IJCAI*, 2015.

[14] Zhe Wang, Ling-Yu Duan, Tiejun Huang, and Wen Gao, "Affinity preserving quantization for hashing: A vector quantization approach to learning compact binary codes," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI Press, 2016, pp. 1102–1108.

[15] Zhe Wang, Ling-Yu Duan, Junsong Yuan, Tiejun Huang, and Wen Gao, "To project more or to quantize more: Minimizing reconstruction bias for learning compact binary codes," in *IJCAI*, 2016.

[16] Antonio Torralba, Robert Fergus, and Yair Weiss, "Small codes and large image databases for recognition," *In Proceedings of CVPR*, 2008.

[17] Weihao Kong and Wu-Jun L, "Double-bit quantization for hashing," *In Proceedings of the Twenty-Sixth AAAI*, 2012.

[18] Mohammad Norouzi, Ali Punjani, and David Fleet, "Fast exact search in hamming space with multi-index hashing," *CVPR*, 2012.

[19] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," *CVPR*, 2008.

[20] R. Salakhutdinov and G. Hinton, "Semantic hashing," *Int. J. Approx. Reasoning*, 2009.

[21] M. Minsky and S. Papert, "Perceptrons," *MIT Press*, 1969.

[22] D. Greene, M. Parnas, and F. Yao, "Multi-index hashing for information retrieval," *IEEE FOCS*, 1994.

[23] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," *ACM STOC*, 1998.

[24] Alexandr Andoni and Piotr Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *In IEEE FOCS*, 2006.

[25] Mani Malek Esmaeili, Rabab Kreidieh Ward, and Mehrdad Fatourechi, "A fast approximate nearest neighbor search algorithm in the hamming space," *PAMI*, 2012.

[26] Eng-Jon Ong and Miroslaw Bober, "Improved hamming distance search using variable length substrings," *CVPR*, 2016.

[27] Zhe Wang, Ling-Yu Duan, Jie Lin, Tiejun Huang, Wen Gao, and Miroslaw Bober, "Component hashing of variable-length binary aggregated descriptors for fast image search," in *Image Processing (ICIP), 2014 IEEE International Conference on*. IEEE, 2014, pp. 2217–2221.

[28] Ling-Yu Duan, Jie Lin, Zhe Wang, Tiejun Huang, and Wen Gao, "Weighted component hashing of binary aggregated descriptors for fast visual search," *IEEE Transactions on multimedia*, vol. 17, no. 6, pp. 828–842, 2015.

[29] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg, "Searching in one billion vectors: re-rank with source coding," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2011, pp. 861–864.

[30] Antonio Torralba, Rob Fergus, and William T Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *PAMI*, 2008.