Fast Approximate Nearest Neighbor Search in Hamming Space Leveraged by Indexing in Euclidean Space

Bin Fan[a,b,c], Baoqian Zhang[a,d], Qingqun Kong[b,c,*], Zhiheng Wang[e,*], Jiwen Lu[f], Chunhong Pan[a,b]

[a] *National Laboratory of Pattern Recognition, China*
[b] *Institute of Automation, Chinese Academy of Sciences, China*
[c] *University of Chinese Academy of Sciences, China*
[d] *China Foreign Affairs University, China*
[e] *Henan Polytechnic University, China*
[f] *Tsinghua University, China*

Abstract

Fast approximate nearest neighbor search has been well studied for real-valued vectors, however, the methods for binary descriptors are less developed. The paper addresses this problem by resorting to the well established techniques in Euclidean space. To this end, the binary descriptors are firstly mapped into low dimensional float vectors under the condition that the neighborhood information in the original Hamming space could be preserved in the mapped Euclidean space as much as possible. Then, KD-Tree is used to partitioning the mapped Euclidean space in order to quickly find approximate nearest neighbors for a given query point. This is identical to filter out a subset of nearest neighbor candidates in the original Hamming space due to the property of neighborhood preserving. Finally, Hamming ranking is applied to the small number of candidates to find out the approximate nearest neighbor in the original Hamming space, with only a fraction of running time compared to the bruteforce linear scan. Our experiments demonstrate that the proposed method significantly outperforms the state of the art, improving the searching accuracy by 30% (from

*Corresponding authors
Email addresses: bfan@nlpr.ia.ac.cn (Bin Fan), qingqun.kong@ia.ac.cn (Qingqun Kong), wzhenry@eyou.com (Zhiheng Wang)

60% to 90%) when the searching speed maintains two orders of magnitude faster than the linear scan for a one million database.

Keywords: Approximate Nearest Neighbor Search, Binary Descriptors, KD-Tree, Feature Matching, Locality Preserving Projection

1. Introduction

Finding nearest neighbor of high-dimensional features against a large scale database is of critical importance in pattern recognition and computer vision applications, such as image retrieval [1, 2], data clustering [3, 4], structure from motion [5, 6], and so on [7, 8, 9]. Since the exhaustive linear scan of nearest neighbor has linear complexity to the size of dataset, it is extremely computational expensive for large scale databases. As a result, researchers have proposed various approximate nearest neighbor searching (ANN) methods with sublinear complexity to overcome this limitation. These methods, such as KD-Tree and its variants [10, 11, 7], vocabulary tree [9], hashing [12, 13, 14, 15], usually perform well with real-valued features, enabling very fast approximate nearest neighbor search for various local descriptors like SIFT [8], LIOP [16], L2Net [17], and so on [18, 19, 20, 21, 22]. To ensure a good trade-off between precision and matching speed, the dimension of the considered real-valued features should not be too high.

In addition to real-valued features, representing image data or local patches as binary codes (binary descriptors) has gaining growing interest. This is because that binary descriptors are storage efficient and their Hamming distance can be computed very fast by several machine instructions. These advantages facilitate the fast development of binary descriptors, both in terms of hand-crafted features (BRISK [23], FREAK [24], FRIF [25], etc.) and learning based ones (e.g., ORB [26], RFD [27], BinBoost [28], LDB [29]). Although computing Hamming distance of binary descriptors is substantially faster than computing Euclidean distance of real-valued descriptors, it remains too slow when it has to match millions of such descriptors which is often the case for many applications

nowadays. On the other hand, directly applying the ANN methods designed for real-valued descriptors to the binary ones is either inapplicable or less effective (i.e, the performance will be severely degraded) [30, 31]. Therefore, it is highly desirable to develop specific ANN methods for binary descriptors.

Compared to the ANN methods for real-valued descriptors, little attention has been paid to the ANN methods for binary descriptors. The most commonly used method [31] for this purpose is adapted from the well-known Locality Sensitive Hashing (LSH) [13] by randomly taking several bits of the binary descriptor as the hash key. This is analog to the hashing functions used when dealing with real-valued descriptors by hashing techniques. Two widely used hashing strategies, multiple tables and multi-probes [32], are used to improve the performance as well. Another well known ANN method for binary descriptors is the hierarchical clustering trees (HCT) [33], which divides the database hierarchically by randomly selecting binary descriptors. These two kinds of ANN methods have been incorporated into the famous FLANN library [34] and have been the primary choices for fast approximate nearest neighbor searching in Hamming space. However, they are still less efficient (both in terms of speed and accuracy) as we will show in our experiments.

In this paper, we propose a fast approximate nearest neighbor searching method for binary descriptors by resorting to the well developed ANN methods in Euclidean space, so as to further improve the matching speed of the state of the art without loss of the searching precision. Previous work on ANN has demonstrated that converting real-valued features to binary ones (i.e., hashing technique) could accelerate the matching speed of real-valued features, while in this paper, we will demonstrate that projecting binary descriptors to real-valued ones is also helpful for fast approximate nearest neighbor searching in Hamming space. The core idea of our method is based on two observations from previous methods: (1) The ANN methods in Euclidean space such as KD-Tree are highly reliable when the dealt real-valued features are with low dimension. (2) The key of fast approximate nearest neighbor search is to quickly find out a small subset of candidates containing the true nearest neighbor with high probabil-

ity. Motivated by these, our method first projects the binary descriptors into low-dimensional real-valued vectors by preserving their neighboring relationship as much as possible (e.g., using Locality Preserving Projections [35]), and then uses the KD-Tree (a typical ANN method in Euclidean space) to quickly find out a small number of candidates for a given query point. Finally, the Hamming distances of all candidates to the query point are computed and sorted to find out the nearest neighbor. In other words, our ANN method in Hamming space is leveraged by using KD-Tree in Euclidean space. As we will show in the experiments, this method significantly improves the currently popular FLANN library with much larger accelerating rate at the same precision. The success of our method can be attributed into two factors. First, since the purpose of using KD-Tree to the projected low-dimensional real-valued features is to return a number of candidates that are further checked as nearest neighbor by Hamming ranking, the projected real-valued features are not required to preserve the exact neighborhood information with high confidence. On the contrary, it is only loosely required to have the true nearest neighbor among the returned candidates by fast searching in Euclidean space. For this purpose, learning a locality preserving projection from the dataset is not a hard task. Second, applying KD-Tree to low-dimensional vectors is highly reliable so that the risk of incorrect candidates returned by searching KD-Tree is minimal. Therefore, our method smartly combines two existing techniques (Locality Preserving Projection and KD-Tree) to achieve a novel and effective usage in fast nearest neighbor search of high dimensional binary descriptors, for which each of the two techniques performs quite poor. Due to these properties, we term our method as Binary Neighborhood Preserving KD-Tree (BNP-KDTree). It is worthy to point out that the proposed method is a framework for conducting fast approximate nearest neighbor search in Hamming space through going back to the Euclidean space, and so any ANN method in Euclidean space can replace the KD-Tree used in this paper.

The rest of this paper is organized as follows. Section 2 introduces the related work on fast nearest neighbor search. Section 3 elaborates our method

in detail with analysis and discussions about its properties. Then, experiments are conducted on Section 4 to show the effectiveness of the proposed method as well as comparison to the state of the art. Conclusions are drawn in Section 5.

2. Related Work

2.1. Fast Nearest Neighbor Search in Euclidean Space

The methods for fast nearest neighbor search in Euclidean space can be mainly divided into two categories, one is tree based method while the other is on the basis of building hash tables. KD-Tree [10, 11, 1] is perhaps the most popular data structure for indexing real-valued vectors, which is essentially a form of balanced binary tree. Each node of the tree splits the dataset into two parts according to a threshold value at a selected dimension. The dimension is usually selected as the one with the greatest variance in the data points belong to this node, and the nodes are iteratively added into a KD-Tree until the sizes of leaf nodes are less than a predefined number. KD-tree has been widely used in computer vision community for various applications, such as object recognition [8, 3], image retrieval [2], shape indexing [1], structure from motion [6]. Silpa-Anan and Hartley [11] proposed randomization techniques to make the searches in multiple KD-Trees as independent as possible so as to improve its performance when dealing with high dimensional descriptors, such as the 128-d SIFT descriptors. Instead of the hyperplane defined by a coordinate axis, Jia et al. [7] proposed to partition the data space by hyperplanes that are linear combination of several coordinate axes. While KD-Tree and its variants partition the data space by hyperplanes, another kind of tree structure uses clustering algorithm to decompose the data space. Typical methods of this kind include hierarchical k-means tree [36], spill tree [37], cover tree [38], vocabulary tree [9], ball tree [3]. Muja and Lowe [31] presented a wide range of comparisons to show that the multiple randomized KD-Trees proposed in [11] is the most effective one for matching high dimensional descriptors. They also

proposed a priority search k-means tree, which performs on par with the multiple randomized KD-Trees in some cases.

Hashing is also a very famous technique for nearest neighbor searching problem. It converts real-valued vectors into binary codes, which are used to build hash tables. Then, data points falling into the same or nearby hash buckets to the query point are taken out as candidate points, from which the nearest neighbor is obtained by ranking their distances. This technique has sublinear complexity since it only needs to compute a small number of distances compared to the exhaustive linear scan. How to obtain the hashing functions used to generate binary codes is at the core of this kind of method. Locality Sensitive Hashing (LSH) [13] generates random projections from Gaussian distributions as hashing functions so as to map similar data points to the same bucket with high probability. It usually needs longer hash codes to achieve satisfactory performance. Multi-probe LSH [32] improves LSH by retrieving data points from nearby hash buckets, enabling higher accuracy with lower storage. Cheng et al. [5] applied the idea of cascade to LSH and proposed the CasHash for fast nearest neighbor search. To obtain better performance with smaller codes, many researchers have proposed to learn data-dependent hashing functions from labeled data. Liu et al. [12] proposed KSH to learn hashing functions by optimizing over the pairwise labels with kernels. Shen et al. [14] proposed SDH to learn hashing functions by directly applying discrete optimization to the hashing problem. Maximizing the kNN classification [39] or ranking performance [40, 41] has also been considered in learning hashing functions. Deep learning has been recently used to learn hashing functions, such as CNNH [42], DPSH [43], HashNet [15], HashGAN [44], and significantly outperforms the traditional learning to hash methods. The hashing based nearest neighbor searching methods can only use very small number of hash codes, otherwise, the huge storage requirement for the corresponding hash table will make these methods impractical.

2.2. Fast Nearest Neighbor Search in Hamming Space

Due to the efficiency in storage and computation of Hamming distance, binary descriptors have gained fast development in the past years. However, their fast nearest neighbor search methods are less developed. Designing tree-based data structures is also a prominent solution for this problem. Muja et al. [31] modified the hierarchical k-means tree [36] to the hierarchical clustering tree (HCT) so as to handle binary vectors. HCT hierarchically selects random binary codes from the database to form the tree nodes and uses them to cluster data points in the database. Ma et al. [45] improved HCT by selecting distinctive bits to construct the clustering trees. Feng et al. [46] resorted to use random trees by supervised learning of nodes which make trees have uniform leaf size and low error rates.

Indexing by hash tables is another solution for fast search in Hamming space. The most common way is to randomly select a few bits as hash key, which is often called as LSH in the context of binary descriptor [31, 46, 47]. This method is less accurate due to the thick boundary in Hamming space [30]. To improve the searching accuracy and maintain the high efficiency, Esmaeili et al. [47] proposed the error weighted hashing (EWH) by checking the bucket distances between the retrieved candidates and the query point. Compared to LSH, EWH can filter out a large number of candidate neighbors returned by hash buckets so as to accelerate the searching speed. Feng et al. [48] proposed to select hashing bits with uniform hash buckets and high collision rates as much as possible. Norouzi et al. [49] proposed the multi-index hashing (MIH) for fast exact nearest neighbor search. MIH builds multiple hash tables for non-overlapping substrings of binary codes respectively. All the neighbors within a radius to the query point can be quickly searched by retrieving the hash buckets with much smaller radius on these tables independently. Eghbali et al.[50] extended MIH to the online setting where the database is dynamic growing. The key of these methods is to probe hash buckets within some radius around the hash key of the query point. However, for high dimensional binary descriptors, most buckets are empty and the nearest Hamming distance is usually large. Therefore, to guarantee the

searching precision, it often requires the searching radius being large enough to cover the nearest neighbor, in which case, the number of probed buckets even exceeds the database size and so the speed could be even slower than linear scan.

Our method is specially designed for the fast Hamming search problem of high dimensional binary descriptors. It projects the high dimensional binary descriptors into low dimensional float vectors, which could to some extent preserve the neighborhood information of the binary descriptors and are used to construct KD-Tree for fast retrieving of a few candidates. The Hamming distances between these candidates to the query point are computed and sorted to find the nearest neighbor. In other words, our solution for fast Hamming search is aided by the reliable fast search technique in Euclidean space (e.g., KD-Tree), which is well motivated and significantly different from previous methods.

3. The Proposed Method

3.1. Overview

The problem we solved is to fast indexing a database of high dimensional binary descriptors. Fig. 1 briefly depicts the proposed solution. In the train stage (also known as database construction), we need to construct a data structure to partitioning the high dimensional Hamming space spanned by the database. For this purpose, we first project the binary descriptors into low dimensional float vectors by linear projections. These linear projections have to be capable of preserving the neighborhood information of the original binary descriptors since our target is for nearest neighbor searching. Due to this reason, we use the Locality Preserving Projections (LPP) [35] in this paper. LPP is an unsupervised method for learning linear projections to preserve the locality information as much as possible in the projected space. Then, the standard KD-Tree [10], which is an advanced method for fast indexing of low dimensional float vectors, is applied to the projected vectors to form a partition of them. Such a partition in the low dimensional Euclidean space is essentially a partition

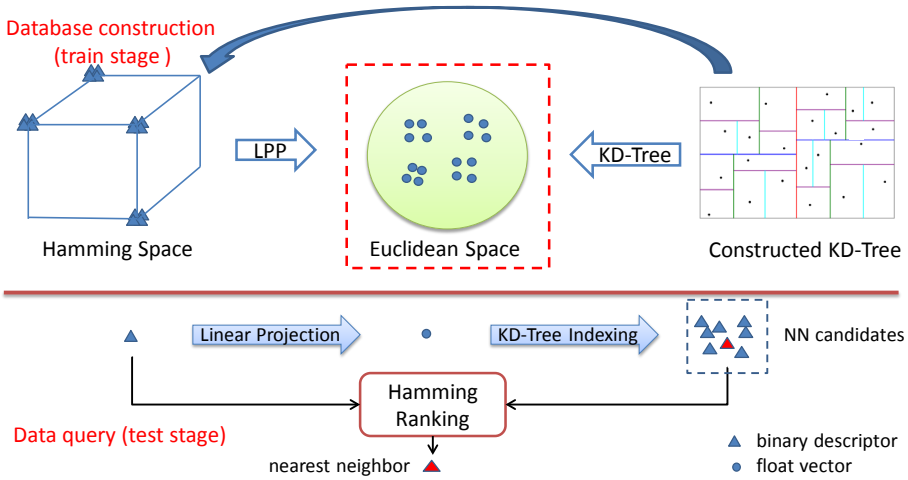

Figure 1: Overview of the proposed method for fast Hamming space search. The upper part depicts the train stage while the bottom part illustrates the data query stage. The low dimensional float vectors in the dash red lines are intermediate vectors used for constructing KD-Tree, and will be discarded for further use. The data space partitioned by the constructed KD-Tree in the Euclidean space corresponds to a specific partition of the database, which is useful in quickly filtering out a small set of candidates in the query stage. See text for more details.

in the original binary descriptors since all the projected vectors are generated from the original high dimensional binary descriptors. In the test stage (i.e., data query), we are given a query binary descriptor. The query descriptor is first projected into a low dimensional float vector through linear projections learned by LPP in the train stage. The projected float vector is then used to retrieve a set of candidate binary descriptors in the database by traversing the constructed KD-Tree in the train stage. Finally, the Hamming distances between the query descriptor and the candidate descriptors are computed and sorted to obtain the nearest neighbor of the query data. The described procedures for database construction and data query are outlined in Algorithm 1 and Algorithm 2 respectively. Since KD-Tree indexing can quickly filter out a small number of candidates, the proposed method only needs to compute a few

Algorithm 1 *BNP-KDTree: Database Construction*

Input:

A set of binary features $b_1, b_2, \cdots, b_n$, dimension of the projected float vectors k

Output:

k linear projections: $\alpha_1, \alpha_2, \cdots, \alpha_k$, and constructed KD-Tree $\mathbb{K}$

1: Compute the locality preserving projections by solving the eigenvector problem in Equ. (2), obtaining the k linear projections: $\alpha_1, \alpha_2, \cdots, \alpha_k$.

2: Apply the obtained linear transformation $A = [\alpha_1 \alpha_2 \cdots \alpha_k]$ to the dataset of binary features, obtaining a set of k-dimensional float vectors $x_1, x_2, \cdots, x_n$, where $x_i = A^T b_i$

3: Build a KD-Tree $\mathbb{K}$ for $x_1, x_2, \cdots, x_n$ according to Section 3.3.

4: **Output**: $\alpha_1, \alpha_2, \cdots, \alpha_k$, and one KD-Tree $\mathbb{K}$.

Hamming distances when searching for the nearest neighbor in the database. Therefore, our method can query binary descriptors very efficiently.

In the following subsections, we will describe in detail of the two components of our method and then give analysis about its properties.

3.2. Locality Preserving Projection for Binary Features

Given a set of m-bits binary features $b_1, b_2, \cdots, b_n \in \{1, -1\}^m$, our target is to find $k << m$ linear projections $\alpha_1, \alpha_2, \cdots, \alpha_k \in \mathcal{R}^m$ such that the projected float vectors $x_1, x_2, \cdots, x_n \in \mathcal{R}^k$ preserve the neighboring relationships of the binary features. Here, $x_i = A^T b_i$, where $A = [\alpha_1 \alpha_2 \cdots \alpha_k] \in \mathcal{R}^{m \times k}$. The neighborhood information of a binary feature b_i can be denoted as $w_{ij} \in \{0, 1\}$, where $w_{ij} = 1$ means b_j is within the neighborhood of b_i. All these neighboring information of $b_1, \cdots, b_n$ constructs the affinity matrix $W \in \{0, 1\}^{n \times n}$ of the dataset. In Hamming space, the neighborhood of a binary feature is usually defined either by a distance threshold ϵ or simply as K nearest features. In this paper, we use the former one to define the neighbors. Note that we do not assign different weights to neighbors since our purpose is just to preserve

Algorithm 2 *BNP-KDTree: Data Query*

Input:

a query binary descriptor b, k linear projections: $\alpha_1, \alpha_2, \cdots, \alpha_k$, the constructed KD-Tree $\mathbb{K}$, number of returned candidates Q

Output:

nearest neighbor b'

1: Compute the low dimensional float vector of the query descriptor by $x = [\alpha_1 \alpha_2 \cdots \alpha_k]^T b$.

2: Use x to traverse the KD-Tree $\mathbb{K}$ and return Q candidate data points in the database.

3: Compute Q Hamming distances between the query data b and the returned candidates respectively.

4: Sort the Q distances and take the candidate point with the smallest distance as the nearest neighbor of b, denoted as b'.

5: **Output**: nearest neighbor b' of the query data b.

the neighboring relationships after projection, while their relative orders are not important. Such an objective is somehow easier to achieve than preserving the ranking orders among the preserved neighborhood information. To this end, the objective function can be formulated as:

$$\mathcal{L} = \sum_{ij} \|x_i - x_j\|^2 w_{ij} = \sum_{ij} \left\|A^T b_i - A^T b_j\right\|^2 w_{ij} \tag{1}$$

Minimizing $\mathcal{L}$ will impose penalties to the neighboring points that have large distances after projection. This problem has been well addressed as Locality Preserving Projections (LPP) [35] in the area of dimension reduction/feature extraction. Through simple algebra formulation, the analytical solution of minimizing Equ. (1) can be obtained by solving the following generalized eigenvector problem:

$$BLB^T A = \lambda BDB^T A \tag{2}$$

where $B = [b_1, b_2, \cdots, b_n]$ is the data matrix, $L = D - W$ is the Laplacian matrix of W, and D is a diagonal matrix whose entries are row sums of W, i.e., $D_{ii} =$

$\sum_{j=1}^{n} w_{ij}$. The k projections $\alpha_1, \alpha_2, \cdots, \alpha_k$ are the eigenvectors corresponding to the k smallest eigenvalues respectively.

3.3. Fast Indexing of Low Dimensional Projected Float Vectors with KD-Tree

Given n low dimensional float vectors $x_1, x_2, \cdots, x_n$ obtained by projecting the high dimensional binary descriptors as described in Section 3.2, the next step of our method is to partitioning the space spanned by $x_1, x_2, \cdots, x_n$ with KD-Tree so as to enable fast indexing of nearest neighbor in the projected low dimensional space. Since each data point x_i contained in the constructed KD-Tree is generated from a high dimensional binary descriptor b_i, the constructed KD-Tree actually partitions the original database of binary descriptors $b_1, b_2, \cdots, b_n$. As a result, through querying a low dimensional float vector x that is obtained by linear projecting a high dimensional binary descriptor b in the constructed KD-Tree, we can quickly get a small set of data points in database. Due to the property of KD-Tree, the projected vectors of these data points are nearest neighbor candidates of x in the low dimensional Euclidean space. On the other hand, since the linear projection can preserve neighborhood information to some extent, these data points in database could also be served as nearest neighbor candidates in the original Hamming space. Based on these evidences, linear scan among these candidates is finally conducted to return the nearest neighbor of the query binary descriptor. Due to the small number of candidates that have to compute and sort distances, the query speed can be hundreds or thousands faster than the brutefore linear search among the whole database.

The node of a KD-Tree is usually taken as a specific dimension of the data along with a threshold. By comparing the value of this dimension with the threshold, the database is split into two parts, i.e., each node of the KD-Tree has two child nodes. We follow the standard implementation of KD-Tree, for a given node, it computes the variance for each dimension based on the data points belonging to this node. Then, the dimension corresponding to the largest variance is adopted by this node to split its data points into two parts (thus generating two new nodes), using the mean data value of this dimension as

the splitting threshold. Starting from the root node containing all data points in the database, this procedure is iteratively conducted to add new nodes to the KD-Tree until reaching the leaf nodes whose sizes are less than a predefined number T. At the run-time, a query point traverses the constructed KD-Tree by comparing with the thresholds at the selected dimensions until reaching the leaf node. To increase the searching precision at the cost of speed, priority search is used to traverse multiple leaf nodes to contain more candidates. Specifically, each time when reaching a leaf node, the data points contained in this leaf node are taken into the set of candidate nearest neighbors. If the number of candidate nearest neighbors is not enough, descending from the most priority node in the queue to the reached leaf node is conducted to add additional candidates. This procedure is repeated until the number of candidates is no less than the predefined number, which controls the trade-off between searching accuracy and speed. The priority of a node in the queue is decided by its distance to the query point so that the closest branch in the KD-Tree will be explored first.

3.4. Analysis

Complexity: The query time of our method consists of three parts: (1) the time t_1 used for linear projecting the query binary descriptor into a low dimensional float vector, (2) the time t_2 used for indexing KD-Tree with the projected float vector, (3) the time t_3 used for computing and sorting the Hamming distances between the query binary descriptor and those binary points returned by traversing KD-Tree. t_1 requires km multipliers and $k(m-1)$ additions, where k is the number of linear projections and m is the dimension of the binary descriptor. Supposing there are T data points associated with each leaf node of the KD-Tree, retrieving Q candidates needs to descending down the tree $r = \lceil Q/T \rceil$ times, each time of which has to conduct at most h comparisons ($h \leq log2(n)$ is the tree depth, and n is the number of binary descriptors in the database). Conducting these comparisons is fast, however, besides these comparisons, t_2 includes time used for maintaining the priority queue in KD-Tree indexing. This part can not be quantitatively analyzed and often occupies the most of t_2. Ac-

cording to our experiments, t_2 takes $1/4 \sim 1/2$ time of t_3. Supposing there are Q candidates returned by indexing the KD-Tree, t_3 is Q/n of that used in linear scan of the database. Usually, due to the small value of k, t_1 is neglectable, so the total query time is approximately $1.25 \sim 1.5Q/n$ of that used in linear scan. As a result, Q actually controls the searching accuracy and speed-ups over linear scan. The more candidates returned by KD-Tree indexing (i.e., the larger Q is), the higher accuracy will it be. At the meantime, the longer time will it need. According to our experiments, set $Q \approx 0.006n$ could achieve a precision as high as 90% with over 100 times speed-up for querying a database containing one million data points.

The space requirement of our method is similar to that of tree based methods. Firstly, it has to store the binary descriptors as the database and their indexes in the leaf nodes. Then, each non-leaf node in the KD-Tree needs 6 bytes to store the splitting dimension (2 bytes for dimension as high as 65536) and threshold value (4 bytes as it is a float value). For a database with n data points, if each leaf node contains T data points, then $\lceil n/T \rceil$ leaf nodes are enough. For $\lceil n/T \rceil$ leaf nodes, it needs at most $\lceil n/T \rceil$ non-leaf nodes in the tree, so the storage required to store a KD-Tree is at most $6\lceil n/T \rceil$ bytes. For a typical database of one million data points with leaf size set as 50, this storage is about $6 \times 1\text{M}/50 = 120\text{K}$ bytes. Besides this basic space requirement as other KD-Tree based methods, our method needs additionally $4mk$ bytes to store k linear projections, for which several hundreds of KB is enough. Note that the low dimensional float vectors projected from the original binary descriptors are only used to construct the KD-Tree to partitioning the database, they do not need to be stored once the tree has been constructed since all the partition information is efficiently contained in the constructed KD-Tree as we have analyzed before. In summary, our method is scalable to large databases.

Performance: The technical route of our method is identical to existing fast approximate nearest neighbor search methods, i.e., quickly filtering out a small number of candidates and then using linear scan across the candidates to find the nearest neighbor. The searching accuracy and speed is mainly deter-

mined by the quantity of candidates and the probability that they contain the groundtruth nearest neighbor. Good methods should contain the groundtruth nearest neighbor with high probability by as small number of candidates as possible. In our method, KD-Tree is operated in low dimensional Euclidean space, which guarantees its high accuracy with a relatively small number of candidates. Meanwhile, since the purpose of KD-Tree indexing in our method is to find out a subset of candidates containing the true nearest neighbor, mapping from the original high dimensional binary descriptors to the low dimension float vectors does not need to precisely preserve the distance orders. In this case, ensuring the nearest neighbor lies in a small radius of the query point after projection is good enough. This is a relative easy task for LPP[1]. Therefore, all these factors together make our method being highly effective and efficient. We will show in the next section that the proposed method significantly outperforms the state of the art. For example, when querying a database with one million data points, it achieves 100 times speed up over linear scan while still maintaining a precision as high as 90%. For comparison, the most competitive HCT implemented in the FLANN library could only obtain 50 times speed up with the same precision. Finally, it is worthy to point out that the proposed method is a framework that effectively uses the advantages of ANN methods in Euclidean space to solve the ANN problem in Hamming space. Although we use KD-Tree in this paper, any other ANN method can be integrated in our method as well. This is similar for LPP and so any dimensionality reduction method with property of neighborhood preserving can be used in our method.

[1]Actually, most nearest neighbors in the original Hamming space will not be the nearest ones in the mapped Euclidean space. According to our experiments, only about 1% query points have nearest neighbors in the projected Euclidean space that are identical to the ones found in the original Hamming space.

4. Experiments

In our experiments, we use binary descriptors extracted from images of the large scale structure from motion dataset [51]. This dataset contains several subsets, each of which corresponds to one landmark and contains thousands of images collected from the Internet. We use the Madrid subset containing 1344 images. FRIF feature [25] is used to extract keypoints along with their binary descriptors from these images. In total, there are 10,366,480 FRIF keypoints detected in these images, each of which is represented by a 512-bits binary code (FRIF descriptor). We randomly sample 10,000 FRIF descriptors from all descriptors as the query set. From the remaining ones, we randomly sample 100K, 1M, 10M descriptors respectively to form three databases with different sizes.

In our method, 25000 samples are randomly selected from the database to construct the graph laplacian matrix in order to learn the LPP projections. The adjacent edges in graph laplacian matrix are formed by neighboring samples whose Hamming distances are smaller than 175. Binary descriptors are linear projected into 20-d float vectors by LPP. The leaf size of KD-Tree is set as 50. These parameter settings are summarized in Table 1, which are determined by our experiments and we will give a parameter study in the next subsection.

Parameter	Description	Value
k	LPP Dimensionality	20
ϵ	binary neighborhood threshold	175
T	leaf size of the KD-Tree	50

Table 1: Description and default parameter values of the proposed method, BNP-KDTree.

For comparison, HCT and LSH implemented in the popular FLANN library [34] are used as competitors for their state of the art performance in this problem. Their parameters are tuned to give the best performance in our experiments. Specifically, we set the number of divisions in each tree node as 20, the leaf size as 100, and use 8 parallel trees for HCT. While for LSH, the

configuration of multi-tables and multi-probe is used since it has been reported with much better performance than the standard LSH. The probe is set as 2 and we change the number of hashing tables to obtain various operating points with different speeds and precisions. Instead of the projected low-dimensional float vectors, building KD-Tree directly by the original 512-bits binary descriptors (simply treaded as 512D real-valued vectors) is also implemented as a baseline. This method is denoted as Bin-KDTree and uses the same leaf size to our method (BNP-KDTree). By comparing to this baseline, the effectiveness and necessity of applying the locality preserving projection to transform the binary descriptors into float vectors can be illustrated. The performance of different methods is demonstrated by drawing curves of speed-up factors at different precisions. Our experiments are conducted on an Intel i5 2.5GHz CPU with 16GB memory, and only single thread is used.

4.1. Results and Analysis

The results of querying different databases with various sizes are shown in Fig. 2, where the curves of BNP-KDTree, Bin-KDTree and HCT are obtained by setting different numbers of candidates returned by traversing trees. For LSH, we plot the curves with different numbers of hashing tables by fixing hashing key length as 30 bits. These settings lead to a number of operating points (i.e., speed-up factors and precisions) on the curves for each method respectively. Apparently, the proposed method (BNP-KDTree) obtains significant better performance than other evaluated methods. For instance, when querying a database with one million data points, as demonstrated in Fig. 2(b), BNP-KDTree still accelerates linear search over 100 times while keeping the precision as high as 90%, while the best competitor HCT can only reach 50 times speed up. The improved acceleration rate of our method over HCT is even more significant for moderate precisions. We can see from Fig. 2(b) that for precision of 70%∼80%, BNP-KDTree has speed-up factors between 300 and 700, while HCT is only about 80∼90 times faster than linear search. Further reducing the searching precision (by decreasing the number of returned candidates), HCT

can not have significant higher speed ups and seems to be saturated in running time. This demonstrates the high basic operation time for HCT to maintain and traverse its tree data structure, which limits the maximal speed ups that HCT can achieve. LSH is less competitive than other methods, and is even worse than directly applying KD-Tree to the binary descriptors (i.e., Bin-KDTree). This is because that the hashing key length is too small compared to the binary feature dimension (30 vs. 512), resulting in a very low collision probability of the generated hashing buckets. Therefore, the candidates returned by probing the nearby hashing buckets will have a low probability containing the true nearest neighbors even though there are many candidates. This is reflected on the curves of Fig. 2, where LSH is less efficient and with low precision. What is worse, for high dimensional binary descriptors, this problem of LSH can not be addressed by simply increasing the hashing key length since it will require a very huge amount of memory to store hashing tables for long key length, thus is impractical. As expected, directly applying KD-Tree to the binary descriptors also does not produce good results since KD-Tree is specifically designed for real-valued vectors. On the contrary, binary descriptors only have two possible values in each dim, making the space division by tree nodes is less meaningful. Different from previous methods, our method smartly converts binary descriptors into float vectors and maintains similar neighborhood relationship in the transformed Euclidean space as that in the original Hamming space. By this way, we conquer the problem of applying KD-Tree to binary descriptors, relying on KD-Tree division in the transformed Euclidean space to obtain a specific partition of the database. As demonstrated by the curves of BNP-KDTree in Fig. 2, such kind of space division is highly effective and leads to very good performance in terms of both efficiency and effectiveness. These observations of different methods are consistent for various database sizes as clearly shown from Fig. 2(a) to Fig. 2(c).

It is also interesting to note that the database size has a positive effect on the acceleration rate over linear scan, which can be seen from Fig. 2(d). The larger the database is, the higher speed up factor a method can achieve, either for

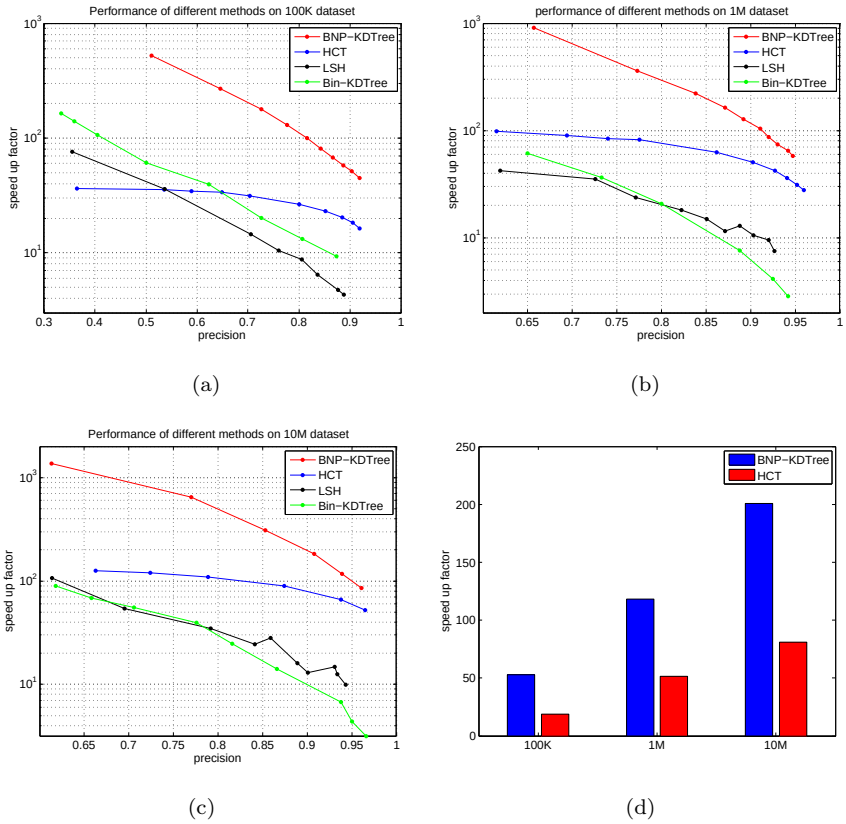

Figure 2: (a)-(c) Accelerations over linear scan with respect to different searching precisions for different methods with various database sizes, i.e., 100K, 1M, 10M. (d) Speed up factors of BNP-KDTree and HCT for different database sizes when precision is kept at 90%.

the proposed BNP-KDTree or the existing HCT. This is because that for small size database, the returned candidates are too few so that the time used for traversing the tree structure can not be neglectable compared to that used for Hamming ranking. While for large size database, the time used for traversing the tree structure only occupies very small fraction of the searching time. Therefore, the speed up is more significant for larger database size. Fig. 2(d) shows the speed up factors of our method and HCT when the searching precision is kept at 90%. Compared to HCT, not only the absolute speed up factors of our method, but also its relative speed improvement along with the increased database size

is more significant, demonstrating the high efficiency of the proposed method.

4.2. Discussions and Parameter Study

In this subsection, we conduct parameter study to show how the performance of our method is related to the different parameter settings. All experiments in this section are conducted on the 1M database and 10,000 queries are used.

Influence of different leaf sizes: Fig. 3(a) shows the approximate nearest neighbor searching performance of our method by using KD-Tree with different leaf sizes. It can be seen that setting leaf size as 50 leads to the best performance. In our method, when indexing the KD-Tree to return a number of candidates, all the data points in a leaf node are either taken in whole as candidates or not. Due to this reason, it actually needs to return larger number of candidates to achieve similar precision for the larger leaf size, thus requiring more processing time. Therefore, larger leaf size leads to worse performance as shown in Fig. 3(a). On the other hand, when the leaf size is too small, it basically corresponds to a over-fined partition of the data space. In this case, it usually needs to traverse more leaf nodes in order to have a high probability containing the true nearest neighbor in the returned candidates. As a result, too small leaf size will have lower speed up factors when the high precision is required, which is the case of red curve (T=25) in Fig. 3(a). When the high precision is not required, smaller leaf size is also more efficient because fewer candidates will be returned and the traversed leaf nodes are not too many as shown in the left part of red curve (T=25) in Fig. 3(a). However, a very small leaf size will lead to significantly over-fined partition, which in turn results in severe degradation of query performance, as clearly demonstrated by the black curve when the leaf size is set as 10 in Fig. 3(a). This property with different leaf sizes is clearly observed in Fig. 3(a), according to which, we set leaf size to be 50 in our method.

Single KD-Tree VS. multiple randomized KD-Trees: In literature, the technique of using multiple randomized KD-Trees has been proposed to increase the performance of high dimensional descriptor matching [11]. However,

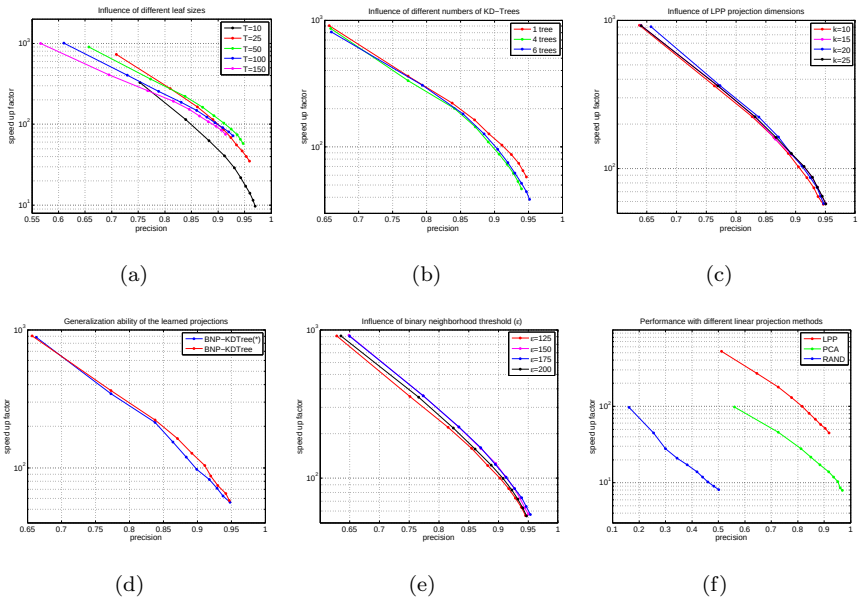

Figure 3: Influences of different parameters. (a) Speed up factors VS. precision for different leaf sizes. (b) Speed up factors VS. precision for different numbers of trees, where the leaf size is set as 50. (c) Influence of the dimension after LPP projection (k). (d) Generalization ability of the learned LPP projections, where (*) means using LPP projections learned from another significantly different dataset. (e) Influence of binary neighborhood threshold (ϵ). (f) Performance with different linear projection methods. Please see text for details.

as shown in Fig. 3(b), using one KD-Tree is highly reliable in our method and leads to higher speed up factors than using multiple randomized KD-Trees at all precisions. This is because that the considered Euclidean space in our method is a relatively low dimensional one, and in this case using multiple randomized KD-Trees can not increase the independence of different searches in the query phase. Meanwhile, the standard KD-Tree with priority search is already very reliable to deal with the low dimensional float vectors. Due to these reasons, it is not necessary to use multiple randomized KD-Trees, which is especially proposed to improve matching performance of high dimensional descriptors. Instead, our method still uses the standard KD-Tree with priority search to deal with the projected low dimensional float vectors. Compared to multiple KD-Trees, using single KD-Tree also requires less memory to store the tree structure and less

time to build the tree.

Influence of LPP dimensions: Fig. 3(c) studies the influence of LPP dimensions, and demonstrates the nearest neighbor searching performance of our method with different LPP dimensions. The performance is relatively stable with different dimensions, perhaps because that our method does not require a restrict neighborhood preserving projection. Among the tested dimensions, 20 is slightly better than other numbers and so we use 20 in our method.

Generalization of the learned LPP projections: Although the LPP projections are learned from a specific dataset, it is basically a transformation from Hamming space to Euclidean space with neighborhood information preserving property. Therefore, applying the learned projections to other datasets should also produce very good performance. To show this point, we replace the learned LPP projections by a new one learned from another dataset, i.e., a set of binary descriptors extracted from images of another structure from motion dataset (Here we use the Gendarmarket subset in [51], which is another landmark different from the one used in our experiments). Except for the LPP projections, other parameters are kept unchanged, and the results are shown in Fig. 3(d). In Fig. 3(d), BNP-KDTree is the result of using LPP projections learned from the same dataset that we conduct nearest neighbor searching, while BNP-KDTree(*) shows the nearest neighbor searching performance on the same dataset with LPP projections learned from another dataset. The images of these two datasets are significantly different from each other since they are captured from two different landmarks. Based on the evidence that BNP-KDTree(*) performs rather similar to BNP-KDTree is Fig. 3(d), we can conclude that the learned LPP projections from a specific dataset can be used in our method to query other datasets as well. In other words, we only need to solve the LPP problem once and the learned projections can be fixed in our method due to its generalization ability.

Influence of binary neighborhood threshold: Fig. 3(e) shows how the binary neighborhood threshold affects the performance of our method. According to this figure, we set the threshold ϵ as 175 although other settings do not

degrade the performance too much.

Effectiveness of the locality preserving projection: We further give a study on how the linear projection affects the ANN searching performance of our method. To show the effectiveness of using locality preserving projection to project the binary descriptors into low dimensional float vectors, we compare with two baselines, one is the random projection and the other is PCA. The results are shown in Fig. 3(f), where RAND means the results obtained by using random linear projections. Obviously, when mapping binary descriptors into low dimensional float vectors, we can not use arbitrary linear projections. In accord with the motivation of our method, only the linear projection methods that are capable of preserving the neighborhood information (at least to some extent) can be used in the proposed framework. Otherwise, the returned candidates by indexing in the low dimensional Euclidean space are nothing to do with the nearest neighbors of the query data points, thus obtaining very bad nearest neighbor searching performance as shown in Fig. 3(f). We simply use the well developed LPP in this paper to demonstrate the effectiveness of the proposed framework as well as to validate the motivation of our method, however, it is natural to replace LPP with any other projection method that has neighborhood preserving ability. Investigating on which one should be the best is out of the scope of this paper.

4.3. Applications to large scale structure from motion

The task of structure from motion is highly depended on the image matching quality. Especially, for large scale structure from motion, the time spent on matching all image pairs of an image collection is usually a bottleneck of the whole procedure since the computational complexity increases in quadratic to the number of images. Thus, the exhaustive feature matching is extremely slow even on the binary descriptors and becomes impractical in this application. We apply our fast approximate nearest neighbor searching method with FRIF binary features to a typical structure from motion application [52] on high resolution or large scale image collections. The results are summarized in

Table 2, where the number of recovered cameras and the time used in feature matching have been reported. The more number of recovered cameras indicates a better performance on estimating the geometries and camera poses from the input images.

Dataset	# images	# recovered images		feature matching time	
		BNP-KDTree	HCT	BNP-KDTree	HCT

Table 2: Reconstruction results of different feature matching methods.

5. Conclusion

Fast nearest neighbor search in Euclidean space has been well studied and is highly reliable for low dimensional float vectors. Based on this observation, we propose to use the approximate nearest neighbor search methods (specifically, we use the well developed KD-Tree technique in this paper) in Euclidean space to solve the same problem in Hamming space, where the fast nearest neighbor search methods are less developed. To this end, we first project the original high dimensional binary descriptors into low dimensional float vectors by the locality preserving projections so as to preserve the neighboring information in the original Hamming space. Then, we use KD-Tree to partitioning the low dimensional float vectors, which is identical to partitioning the original binary database. Finally, given a query binary descriptor, it is projected into a float vector that is then used to indexing the constructed KD-Tree to obtain a small set of candidates. Due to the neighborhood preserving property, such candidates returned by indexing in the low dimensional Euclidean space are very likely to be the neighboring points in the original Hamming space. Thus the Hamming distances between the query data and the candidates are computed and sorted to find out the nearest neighbor of the query data. It is noteworthy that although the components of our method are well established techniques, they are reasonably combined together in this paper to form an elegant solution

towards solving a new problem that neither of them can deal with. Extensive experiments have demonstrated the effectiveness and efficiency of the proposed method.

References

[1] J. S. Beis, D. G. Lowe, Shape indexing using approximate nearestneighbour search in highdimensional spaces, in: IEEE Conference on Computer Vision and Pattern Recognition, 1997, pp. 1000–1006. 2, 5

[2] J. Philbin, O. Chum, M. Isard, J. Sivic, A. Zisserman, Object retrieval with large vocabularies and fast spatial matching, in: IEEE Conference on Computer Vision and Pattern Recognition, 2007, pp. 1–8. 2, 5

[3] B. Leibe, K. Mikolajczyk, B. Schiele, Efficient clustering and matching for object class recognition, in: British Machine Vision Conference, 2006, pp. 789–798. 2, 5

[4] A. K. Jain, Data clustering: 50 years beyond k-means, Pattern Recognition Letters 31 (8) (2010) 651–666. 2

[5] J. Cheng, C. Leng, J. Wu, H. Cui, H. Lu, Fast and accurate image matching with cascade hashing for 3D reconstruction, in: IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 1–8. 2, 6

[6] S. Agarwal, Y. Furukawa, N. Snavely, I. Simon, B. Curless, S. M. Seitz, R. Szeliski, Building Rome in a day, Communications of the ACM 54 (10) (2011) 105–112. 2, 5

[7] Y. Jia, J. Wang, G. Zeng, H. Zha, X.-S. Hua, Optimizing kdtrees for scalable visual descriptor indexing, in: IEEE Conference on Computer Vision and Pattern Recognition, 2010, pp. 3392–3399. 2, 5

[8] D. G. Lowe, Distinctive image features from scale-invariant keypoints, International Journal of Computer Vision 60 (2) (2004) 91–110. 2, 5

[9] D. Nistér, H. Stewénius, Scalable recognition with a vocabulary tree, in: IEEE Conference on Computer Vision and Pattern Recognition, 2006, pp. 2161–2168. 2, 5

[10] J. L. Bentley, Multidimensional binary search trees used for associative searching, Communications of the ACM 18 (9) (1975) 509–517. 2, 5, 8

[11] C. Silpa-Anan, R. Hartley, Optimised KD-trees for fast image descriptor matching, in: IEEE Conference on Computer Vision and Pattern Recognition, 2008, pp. 1–8. 2, 5, 20

[12] W. Liu, J. Wang, R. Ji, Y. Jiang, S. Chang, Supervised hashing with kernels, in: IEEE Conference on Computer Vision and Pattern Recognition, 2012, pp. 2074–2081. 2, 6

[13] A. Gionis, P. Indyk, R. Motwani, Similarity search in high dimensions via hashing, in: International Conference on Very Large Data Bases, 1999, pp. 518–529. 2, 3, 6

[14] F. Shen, C. Shen, W. Liu, H. T. Shen, Supervised discrete hashing, in: IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 37–45. 2, 6

[15] Z. Cao, M. Long, J. Wang, P. S. Yu, HashNet: Deep learning to hash by continuation, in: International Conference on Computer Vision, 2017, pp. 5608–5617. 2, 6

[16] Z. Wang, B. Fan, F. Wu, Local intensity order pattern for feature description, in: International Conference on Computer Vision, 2011, pp. 603–610. 2

[17] Y. Tian, B. Fan, F. Wu, L2Net: Deep learning of discriminative patch descriptor in euclidean space, in: IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 6128–6136. 2

[18] H. Bay, A. Ess, T. Tuytelaars, L. V. Gool, SURF: Speeded up robust features, Computer Vision and Image Understanding 110 (3) (2008) 346–359. 2

[19] K. M. Yi, E. Trulls, V. Lepetit, P. Fua, LIFT: Learned invariant feature transform, in: European Conference on Computer Vision, 2016, pp. 467–483. 2

[20] K. Simonyan, A. Vedaldi, A. Zisserman, Learning local feature descriptors using convex optimisation, IEEE Transaction on Pattern Analysis and Machine Intelligence 36 (8) (2014) 1573–1585. 2

[21] Z. Wang, B. Fan, G. W. an Fuchao Wu, Exploring local and overall ordinal information for robust feature description, IEEE Transactions on Pattern Analysis and Machine Intelligence 38 (11) (2016) 2198–2211. 2

[22] B. Fan, F. Wu, Z. Hu, Rotationally invariant descriptors using intensity order pooling, IEEE Transaction on Pattern Analysis and Machine Intelligence 34 (10) (2012) 2031–2045. 2

[23] S. Leutenegger, M. Chli, R. Siegwart, BRISK: Binary robust invariant scalable keypoints, in: International Conference on Computer Vision, 2011, pp. 2548–2555. 2

[24] A. Alahi, R. Ortiz, P. Vandergheynst, FREAK: Fast retina keypoint, in: IEEE Conference on Computer Vision and Pattern Recognition, 2012, pp. 510–517. 2

[25] Z. Wang, B. Fan, F. Wu, FRIF: Fast robust invariant feature, in: British Machine Vision Conference, 2013. 2, 16

[26] E. Rublee, V. Rabaud, K. Konolige, G. Bradski, ORB: an efficient alternative to SIFT or SURF, in: International Conference on Computer Vision, 2011, pp. 2564–2571. 2

[27] B. Fan, Q. Kong, T. Trzcinski, Z. Wang, C. Pan, P. Fua, Receptive fields selection for binary feature description, IEEE Transactions on Image Processing 23 (6) (2014) 2583–2595. 2

[28] T. Trzcinski, M. Christoudias, V. Lepetit, Learning image descriptors with boosting, IEEE Transactions on Pattern Analysis and Machine Intelligence 37 (3) (2015) 597–610. 2

[29] X. Yang, T. Cheng, Local difference binary for ultra-fast and distinctive feature description, IEEE Transaction on Pattern Analysis and Machine Intelligence 36 (1) (2014) 188–194. 2

[30] T. Trzcinski, V. Lepetit, P. Fua, Thick boundaries in binary space and their influence on nearest-neighbor search, Pattern Recognition Letters 33 (16) (2012) 2173–2180. 3, 7

[31] M. Muja, D. G. Lowe, Scalable nearest neighbor algorithms for high dimensional data, IEEE Transactions on Pattern Analysis and Machine Intelligence 36 (11) (2014) 2227–2240. 3, 5, 7

[32] Q. Lv, W. Josephson, Z. Wang, M. Charikar, K. Li, Multiprobe LSH: Efficient indexing for high-dimensional similarity search, in: International Conference on Very Large Data Bases, 2007, pp. 950–961. 3, 6

[33] M. Muja, D. G. Lowe, Fast matching of binary features, in: Ninth Conference on Computer and Robot Vision, 2012, pp. 404–410. 3

[34] M. Muja, D. G. Lowe, FLANN: Fast library for approximate nearest neighbors, [Online]. http://www.cs.ubc.ca/research/flann. 3, 16

[35] X. He, P. Niyogi, Locality preserving projections, in: Neural Information Processing Systems, 2003, pp. 153–160. 4, 8, 11

[36] K. Fukunaga, P. M. Narendra, A branch and bound algorithm for computing k-nearest neighbors, IEEE Transactions on Computing 24 (7) (1975) 750–753. 5, 7

[37] T. Liu, A. W. Moore, A. Gray, K. Yang, An investigation of practical approximate nearest neighbor algorithms, in: Neural Information Processing Systems, 2004, pp. 825–832. 5

[38] A. Beygelzimer, S. Kakade, J. Langford, Cover trees for nearest neighbor, in: International Conference on Machine Learning, 2006, pp. 97–104. 5

[39] K. Ding, C. Huo, B. Fan, C. Pan, kNN hashing with factorized neighborhood representation, in: International Conference on Computer Vision, 2015, pp. 1098–1106. 6

[40] J. Wang, J. Wang, N. Yu, S. Li, Order preserving hashing for approximate nearest neighbor search, in: ACM International Conference on Multimedia, 2013, pp. 133–142. 6

[41] L. Liu, L. Shao, F. Shen, M. Yu, Discretely coding semantic rank orders for supervised image hashing, in: IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 5140–5149. 6

[42] R. Xia, Y. Pan, H. Lai, C. Liu, S. Yan, Supervised hashing for image retrieval via image representation learning, in: AAAI Conference on Artificial Intelligence, 2014, pp. 2156–2162. 6

[43] W. Li, S. Wang, W. Kang, Feature learning based deep supervised hashing with pairwise labels, in: International Joint Conference on Artificial Intelligence, 2016, pp. 1711–1717. 6

[44] Y. Cao, M. Long, B. Liu, J. Wang, HashGAN: Deep learning to hash with pair conditional wasserstein GAN, in: IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 1287–1296. 6

[45] Y. Ma, H. Xie, Z. Chen, Q. Dai, Y. Huang, G. Ji, Fast search of binary codes with distinctive bits, in: Pacific Rim Conference on Multimedia, 2014, pp. 274–283. 7

[46] Y. Feng, L. Fan, Y. Wu, Fast localization in large-scale environments using supervised indexing of binary features, IEEE Transactions on Image Processing 25 (1) (2016) 343–358. 7

[47] M. Esmaeili, R. Ward, M. Fatourechi, A fast approximate nearest neighbor search algorithm in the hamming space, IEEE Transactions on Pattern Analysis and Machine Intelligence 34 (12) (2012) 2481–2488. 7

[48] Y. Feng, Y. Wu, L. Fan, Real-time SLAM relocalization with online learning of binary feature indexing, Machine Vision and Applications 28 (8) (2017) 953–963. 7

[49] M. Norouzi, A. Punjani, D. Fleet, Fast exact search in hamming space with multi-index hashing, IEEE Transactions on Pattern Analysis and Machine Intelligence 36 (6) (2014) 1107–1119. 7

[50] S. Eghbali, H. Ashtiani, L. Tahvildari, Online nearest neighbor search in binary space, in: IEEE International Conference on Data Mining, 2017, pp. 853–858. 7

[51] K. Wilson, N. Snavely, Robust global translations with 1dsfm, in: European Conference on Computer Vision, 2014, pp. 61–75. 16, 22

[52] C. Wu, Towards linear-time incremental structure from motion, in: International Conference on 3D Vision, 2013, pp. 127–134. 23