Fast Neighborhood Graph Search using Cartesian Concatenation *

Jing Wang[1] Jingdong Wang[2] Gang Zeng[1] Rui Gan[1] Shipeng Li[2] Baining Guo[2]
[1]Peking University [2]Microsoft Research

Abstract

In this paper, we propose a new data structure for approximate nearest neighbor search. This structure augments the neighborhood graph with a bridge graph. We propose to exploit Cartesian concatenation to produce a large set of vectors, called bridge vectors, from several small sets of subvectors. Each bridge vector is connected with a few reference vectors near to it, forming a bridge graph. Our approach finds nearest neighbors by simultaneously traversing the neighborhood graph and the bridge graph in the best-first strategy. The success of our approach stems from two factors: the exact nearest neighbor search over a large number of bridge vectors can be done quickly, and the reference vectors connected to a bridge (reference) vector near the query are also likely to be near the query. Experimental results on searching over large scale datasets (SIFT, GIST and HOG) show that our approach outperforms state-of-the-art ANN search algorithms in terms of efficiency and accuracy. The combination of our approach with the IVFADC system [18] also shows superior performance over the BIGANN dataset of 1 *billion SIFT features compared with the best previously published result.*

1. Introduction

Nearest neighbor (NN) search is a fundamental problem in machine learning, information retrieval and computational geometry. It is also a crucial step in many vision and graphics problems, such as shape matching [13], object retrieval [29], feature matching [8, 35], texture synthesis [23], image completion [16] and so on. Recently, the nearest neighbor search problem attracts more attentions in computer vision because of the popularity of large scale and high-dimensional multimedia data.

The simplest solution to NN search is linear scan, comparing each reference vector to the query vector. The search complexity is linear with respect to both the number of reference vectors and the data dimensionality. Apparently, it is too time-consuming and does not scale well in large scale and high-dimensional problems. Algorithms, including the KD tree [3, 5, 6, 12], BD trees [3], cover tree [7], nonlinear embedding [17] and so on, have been proposed to improve the search efficiency. However, for high-dimensional cases it turns out that such approaches are not much more efficient than linear scan and cannot satisfy the practical requirement. Therefore, a lot of efforts have been turned to approximate nearest neighbor (ANN) search. In the following, we will review representative ANN search algorithms and then present the proposed approach.

1.1. Related work

A lot of algorithms have been developed to solve the ANN search problems. The KD trees [6, 12] have been modified to find ANNs by using the best-first search scheme [3, 5]. Various trees, such as BD tress [3], metric trees [9, 24, 26, 46], hierarchical k-means tree [28], and randomized KD trees [20, 33, 43], have been proposed. FLANN [27] aims to find the best configuration of the hierarchical k-means trees and randomized KD trees, and has been shown to work well in practice.

Locality sensitive hashing (LSH) [10] has been shown to achieve good theory guarantee in finding near neighbors with probability, but it is reported not as good as KD trees in practice [27]. Multi-probe LSH [25] adopts the search algorithm similar to priority search, achieving a significant improvement. Recently, various hashing algorithms are proposed by learning hash functions using metric learning-like techniques, including learnt binary reconstruction [21], kernelized LSH [22], and shift kernel hashing [30], spectral hashing [44], product quantization [18], iterative quantization [14], complementary hashing [45] and order preserving hashing [41].

Recently neighborhood graph search has been attracting a lot of interests [1, 2, 5, 15, 31, 32, 38] because of its simplicity and good search performance. The data structure is a directed graph connecting each vector and its nearest neighbors. The basic procedure of neighborhood graph search starts from one or several seeding vectors, and puts them into a priority queue with the distance to the query being the key. Then the process proceeds by popping the top one in the queue, i.e., the nearest one to the query, and expand-

*This work was done when Jing Wang was an intern at Microsoft Research.

ing its neighborhood vectors (from neighborhood graph), among which the vectors that have not been visited are pushed into the priority queue. This process iterates till a fixed number of vectors are accessed. Using neighborhood vectors of a vector as candidates has two advantages. One is that extracting the candidates is very cheap and only takes $O(1)$ time. The other is that if one vector is close to the query, its neighborhood vectors are also likely to be close to the query. The main research efforts consists of two aspects. One is to build an effective neighborhood graph [1, 31]. The other is to design efficient and effective ways to guide the search in the neighborhood graph, including presetting the seeds created via clustering [31, 32], picking the candidates from KD tress [2], iteratively searching between KD trees and the neighborhood graph [38]. In this paper, we present a more effective way, combining the neighborhood graph with a bridge graph, to search for approximate nearest neighbors.

The inverted index algorithms are widely used for very large datasets of vectors (hundreds of million to billions) due to its small memory cost. Such algorithms usually load the inverted index (and possibly extra codes) into the memory and store the raw features in the disk. A typical inverted index is built by clustering algorithms, e.g., [4, 18, 28, 34, 39], and is composed of a set of inverted lists, each of which corresponds to a cluster of reference vectors. Other inverted indices include hash tables [10], tree codebooks [6] and complementary tree codebooks [37].

1.2. Our approach

In this paper, we propose a new data structure for approximate nearest neighbor search. This structure augments the neighborhood graph with a bridge graph that is able to boost approximate nearest neighbor search performance.

Inspired by the product quantization technology [4, 18], we adopt Cartesian concatenation (or Cartesian product), to generate a large set of vectors, which we call *bridge vectors*, from several small sets of subvectors to approximate the reference vectors. Each bridge vector is then connected to a few reference vectors that are near enough to it, forming a bridge graph. Combining the bridge graph with the neighborhood graph built over reference data vectors yields an augmented neighborhood graph. The ANN search procedure starts by finding the nearest bridge vector to the query vector, and discovers the first set of reference vectors connected to such a bridge vector. Then the search simultaneously traverses the bridge graph and the neighborhood graph in the best-first manner using a shared priority queue.

The advantages of adopting the bridge graph lie in two-fold. First, computing the distances from bridge vectors to the query is very efficient, for instance, the computation for 1000000 bridge vectors that are formed by 3 sets of 100 subvectors takes almost the same time as that for 100 vectors. Second, the best bridge vector is most likely to be very close to true NNs, allowing the ANN search to quickly reach true NNs through bridge vectors. Note that different from applications to fast distance computation [18] and code book construction [4], the goal of Cartesian product in this paper is to build a bridge to connect the query and the reference vectors through bridge vectors.

We evaluate the proposed approach by the feature matching performance on SIFT and HOG features, and the performance of searching similar images over tiny images [36] with GIST features. We show that our approach achieves significant improvements compared with the state-of-the-art in terms of accuracy and search time. We also demonstrate that our approach in combination with the IVFADC system [18] outperforms the state-of-the-art over the BIGANN dataset of 1 billion SIFT vectors [19].

2. Approach

The database $\mathcal{X}$ contains N d-dimensional reference vectors, $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N\}$, $\mathbf{x}_i \in \mathbb{R}^d$. Our goal is to build an index structure using the bridge graph such that, given a query vector $\mathbf{q}$, its nearest neighbors can be quickly discovered. In this section, we first describe the index structure and then show the search algorithm.

2.1. Data structure

Our index structure consists of two components: a bridge graph that connects bridge vectors and their nearest reference vectors, and a neighborhood graph that connects each reference vector to its nearest reference vectors.

Bridge vectors. Cartesian concatenation is an operation that builds a new set out of a number of given sets. Given m sets, $\{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_m\}$, where each set, in our case, contains a set of d_i-dimensional subvectors such that $\sum_{i=1}^m d_i = d$, the Cartesian concatenation of those sets is defined as follows,

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

Here $\mathbf{y}_j$ is a d-dimensional vector, and there exist $\prod_{i=1}^m n_i$ vectors ($n_i = |\mathcal{S}_i|$ is the number of elements in $\mathcal{S}_i$) in the Cartesian concatenation $\mathcal{Y}$. Without loss of generality, we assume that $n_1 = n_2 = \cdots = n_m = n$ for convenience. There is a nice property that identifying the nearest one from $\mathcal{Y}$ to a query only takes $O(dn)$ time rather than $O(dn^m)$, despite that the number of elements in $\mathcal{Y}$ is n^m. Inspired by this property, we use the Cartesian concatenation $\mathcal{Y}$, called bridge vectors, as bridges to connect the query vector with the reference vectors.

Computing bridge vectors. We propose to use product quantization [18], which aims to minimize the distance of each vector to the nearest concatenated center derived from subquantizers, to compute bridge vectors. This ensures that the reference vectors discovered through one bridge vector are not far away from the query and hence the probability that those reference vectors are true NNs is high.

It is also expected that the number of reference vectors that are close enough to at least one bridge vector should be as large as possible (to make sure that enough good reference vectors can be discovered merely through bridge vectors) and that the average number of the reference vectors discovered through each bridge vector should be small (to make sure that the time cost to access them is low). To this end, we generate a large amount of bridge vectors. Such a requirement is similar to [18] for source coding and different from [4] for inverted indices.

Augmented neighborhood graph. The augmented neighborhood graph is a combination of the neighborhood graph $\bar{G}$ over the reference database $\mathcal{X}$ and the bridge graph B between the bridge vectors $\mathcal{Y}$ and the reference vectors $\mathcal{X}$. The neighborhood graph $\bar{G}$ is a directed graph. Each node corresponds to a point $\mathbf{x}_i$, and is also denoted as $\mathbf{x}_i$ for convenience. Each node $\mathbf{x}_i$ is connected with a list of nodes that correspond to its neighbors, denoted by$Adj[\mathbf{x}_i]$.

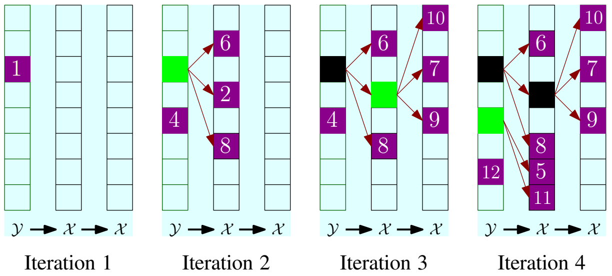
The bridge graph B is constructed by connecting each bridge vector $\mathbf{y}_j$ in $\mathcal{Y}$ to its nearest vectors $Adj[\mathbf{y}_i]$ in $\mathcal{X}$. To avoid expensive computation cost, we build the bridge graph approximately by finding top t (typically 100 in our experiments) nearest bridge vectors for each reference vector and then keeping top b nearest (typically 5 in our experiments) reference vectors for each bridge vector, which is efficient and takes $O(Nt(\log t + b))$ time.

The bridge graph is different from the inverted multi-index [4]. In the inverted multi-index, each bridge vector $\mathbf{y}$ contains a list of vectors that are closer to $\mathbf{y}$ than all other bridge vectors, while in our approach each bridge is associated with a list of vectors that are closer to $\mathbf{y}$ than all other reference data points.

2.2. Query the augmented neighborhood graph

To make the description clear, without loss of generality, we assume there are two sets of n subvectors, $\mathcal{S}_1 = \{\mathbf{y}^1_1, \mathbf{y}^1_2, \cdots, \mathbf{y}^1_n\}$ and $\mathcal{S}_2 = \{\mathbf{y}^2_1, \mathbf{y}^2_2, \cdots, \mathbf{y}^2_n\}$. Given a query $\mathbf{q}$ consisting of two subvectors $\mathbf{q}^1$ and $\mathbf{q}^2$, the goal is to generate a list of T ($T \ll N$) candidate reference points from $\mathcal{X}$ where the true NNs of $\mathbf{q}$ are most likely to lie. This is achieved by traversing the augmented neighborhood graph in a best-first strategy.

We give a brief overview of the ANN search procedure over a neighborhood graph before describing how to make use of bridge vectors. The algorithm begins with a set of (one or several) vectors $\mathcal{P}_s = \{\mathbf{p}\}$ that are contained in the neighborhood graph. It maintains a set of nearest neighbor candidates (whose neighborhoods have not been expanded), using a min-priority queue, which we call the main queue, with the distance to the query as the key. The main queue initially contains the vectors in $\mathcal{P}_s$. The algorithm proceeds by iteratively expanding the neighborhoods in a best-first strategy. At each step, the vector $\mathbf{p}^*$ with top priority (the nearest one to $\mathbf{q}$) is popped from the queue. Then each neighborhood vector in $Adj[\mathbf{p}^*]$ is inserted to the queue if it is not visited, and at the same time it is added to the result set (maintained by a max-priority queue with a fixed length depending on how many nearest neighbors are expected).


1
6
2
4
8
10
6
7
4
9
8
10
6
7
9
8
12
5
11
$\mathcal{Y} \rightarrow \mathcal{X} \rightarrow \mathcal{X}$
$\mathcal{Y} \rightarrow \mathcal{X} \rightarrow \mathcal{X}$
$\mathcal{Y} \rightarrow \mathcal{X} \rightarrow \mathcal{X}$
$\mathcal{Y} \rightarrow \mathcal{X} \rightarrow \mathcal{X}$

Iteration 1 Iteration 2 Iteration 3 Iteration 4

Figure 1. An example illustrating the search process. $\mathcal{Y} \rightarrow \mathcal{X}$: the bridge graph, and $\mathcal{X} \rightarrow \mathcal{X}$: the neighborhood graph. The white numbers are the distances to the query. Magenta denotes the vectors in the main queue, green represents the vector being popped out from the main queue, and black indicates the vectors whose neighborhoods have already been expanded.

To exploit the bridge vectors, we present an extraction-on-demand strategy, instead of fetching all the bridge vectors to the main queue, which leads to expensive cost in sorting them and maintaining the main queue. Our strategy is to maintain the main queue such that it consists of only one bridge vector if available. To be specific, if the top vector $\mathbf{p}^*$ in the main queue is a reference vector, the algorithm proceeds as usual, the same to the above procedure without using bridge vectors. If the top vector is a bridge vector, we first insert its neighbors $Adj[\mathbf{p}^*]$ into the main queue and the result set, and in addition we find the next nearest bridge vector (to the query $\mathbf{q}$) and insert it to the main queue. The pseudo code of the search algorithm is given in Algorithm 1 and an example process is illustrated in Figure 1.

Before traversing the augmented neighborhood graph, we first process the bridge vectors, and compute the distances (the square of the Euclidean distance) from $\mathbf{q}^1$ to the subvectors in $\mathcal{S}_1$ and from $\mathbf{q}^2$ to the subvectors in $\mathcal{S}_2$, and then sort the subvectors in the order of increasing distances, respectively. We denote the sorted subvectors as $\{\mathbf{y}^1_{i_1}, \cdots, \mathbf{y}^1_{i_n}\}$ and $\{\mathbf{y}^2_{j_1}, \cdots, \mathbf{y}^2_{j_n}\}$. As the size n of $\mathcal{S}_1$ and $\mathcal{S}_2$ is typically not large (e.g., 100 in our case), the computation cost is very small (See details in Section 4).

The extraction-on-demand strategy needs to visit the bridge vector one by one in the order of increasing distance from $\mathbf{q}$. It is easily shown that $\operatorname{dist}^2(\mathbf{q}, \mathbf{y}) = \operatorname{dist}^2(\mathbf{q}^1, \mathbf{y}^1) + \operatorname{dist}^2(\mathbf{q}^2, \mathbf{y}^2)$, where $\mathbf{y}$ is consists of $\mathbf{y}^1$ and $\mathbf{y}^2$. Naturally, $\mathbf{y}_{i_1,j_1}$, composed of $\mathbf{y}^1_{i_1}$ and $\mathbf{y}^2_{i_1}$, is the nearest one to $\mathbf{q}$. The multi-sequence algorithm (corresponding to ExtractNextNearestBridgeVector() in Algorithm 1) proposed in [4] is able to fast produce a sequence of pairs

Algorithm 1 ANN search over the augmented neighborhood graph

```
/* $\mathbf{q}$: the query; $\mathcal{X}$: the reference data vectors; $\mathcal{Y}$: the set of bridge vectors; $G$: the augmented neighborhood graph; $Q$: the main queue; $R$: the result set; $T$: the maximum number of discovered vectors; */
Procedure ANNSearch($\mathbf{q}$, $\mathcal{X}$, $\mathcal{Y}$, $G$, $Q$, $R$, $T$)
/* Mark each reference vector undiscovered */
for each $\mathbf{x} \in \mathcal{X}$ do
    Color[$\mathbf{x}$] $\leftarrow$ white;
end for
/* Extract the nearest bridge vector */
$(\mathbf{y}, D) \leftarrow$ ExtractNextNearestBridgeVector($\mathcal{Y}$);
$Q \leftarrow (\mathbf{y}, D)$;
$t \leftarrow 0$
/* Start the search */
while ($Q \neq \emptyset$ && $t \leqslant T$) do
    /* Pop out the best candidate vector and expand its neighbors */
    $(\mathbf{p}, D) \leftarrow Q.\mathrm{pop}()$;
    for each $\mathbf{x} \in Adj[\mathbf{p}]$ do
        if Color[$\mathbf{x}$] = white then
            $D \leftarrow \mathrm{dist}(\mathbf{q}, \mathbf{x})$;
            $Q \leftarrow (\mathbf{x}, D)$;
            Color[$\mathbf{x}$] $\leftarrow$ black; /* Mark it discovered */
            $R \leftarrow (\mathbf{x}, D)$; /* Update the result set */
            $t \leftarrow t + 1$;
        end if
    end for
    /* Extract the next nearest bridge vector if $\mathbf{p}$ is a bridge vector */
    if $\mathbf{p} \in \mathcal{Y}$ then
        $(\mathbf{y}, D) \leftarrow$ ExtractNextNearestBridgeVector($\mathcal{Y}$);
        $Q \leftarrow (\mathbf{y}, D)$;
    end if
end while
return $R$;
```

(i_k, j_l) so that the corresponding bridge vectors are visited in the order of increasing distances to the query $\mathbf{q}$. The algorithm is very efficient and producing the t-th bridge vector only takes $O(\log(t))$ time. Slightly different from extracting a fixed number of nearest bridge vectors once [4], our algorithm automatically determines when to extract the next one, that is when there is no bridge vector in the main queue.

3. Experiments

Setup. We perform our experiments on three large datasets: the first one with local SIFT features, the second one with global GIST features, and the third one with HOG features, and a very large dataset, the BIGANN dataset of 1 billion SIFT features [19].

The SIFT features are collected from the Caltech 101 dataset [11]. We extract maximally stable extremal regions (MSERs) for each image, and compute a 128-dimensional byte-valued SIFT feature for each MSER. We randomly sample $1000K$ SIFT features and $100K$ SIFT features, respectively as the reference and query set. The GIST features are extracted on the tiny image set [36]. The GIST descriptor is a 384-dimensional byte-valued vector. We sample $1000K$ images as the reference set and $100K$ images as the queries. The HOG descriptors are extracted from Flickr images, and each HOG descriptor is a 512-dimensional byte-valued vector. We sample $10M$ HOG descriptors as the reference set and $100K$ as the queries. The BIGANN dataset [19] consists of $1B$ 128-dimensional byte-valued vectors as the reference set and $10K$ vectors as the queries.

Table 1. The parameters of our approach and the statistics. #reference means the number of reference vectors associated with the bridge vectors, and α means the average number of unique reference vectors associated with each bridge vector.

	size	#partitions	#clusters	#reference	α
SIFT	$1M$	4	50	$715K$	11.4%
GIST	$1M$	4	50	$599K$	9.59%
HOG	$10M$	4	100	$5730K$	5.73%

We use the accuracy score to evaluate the search quality. For k-ANN search, the accuracy is computed as r/k, where r is the number of retrieved vectors that are contained in the true k nearest neighbors. The true nearest neighbors are computed by comparing each query with all the reference vectors in the data set. We compare different algorithms by calculating the search accuracy given the same search time, where the search time is recorded by varying the number of accessed vectors. We report the performance in terms of search time vs. search accuracy for the first three datasets. Those results are obtained with 64 bit programs on a $3.4G$ Hz quad core Intel PC with $24G$ memory.

Empirical analysis. The index structure construction in our approach includes partitioning the vector into m subvectors and grouping the vectors of each partition into n clusters. We conduct experiments to study how they influence the search performance. The results over the $1M$ SIFT and $1M$ GIST datasets are shown in Figure 2. Considering two partitions, it can be observed that the performance becomes better with more clusters for each partition. This is because more clusters produce more bridge vectors and thus more reference vectors are associated with bridge vectors and their distances are much smaller. The result with 4 partitions and 50 clusters per partition gets the best performance as in this case the properties desired for bridge vectors described in Section 2.1 are more likely to be satisfied.

Comparison. We compare our approach with state-of-the-art algorithms, including iterative neighborhood graph search [38], original neighborhood graph search (AryaM93) [2], trinary projection (TP) trees [20], vantage point (VP) tree [46], Spill trees [24], FLANN [27], and inverted multi-index [4]. The results of all other methods are obtained by well tuning parameters. We do not report the results from hashing algorithms as they are much worse than tree-based approach, which is also reported in [27, 43]. The neighborhood graphs of different algorithms are the same, and each vector is connected with 20 nearest vectors. We construct approximate neighborhood graphs using the algorithm [42]. Table 1 shows the parameters for our approach,

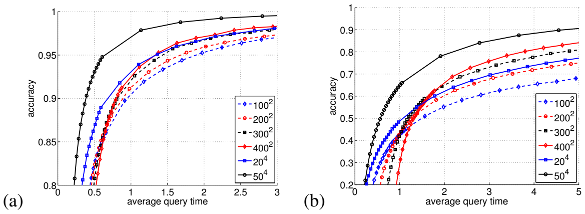


Figure 2. Search performances with different number of partitions and clusters over (a) $1M$ SIFT and (b) $1M$ GIST. x^y: y means #partitions and x is #clusters.

together with some statistics.

The experimental comparisons are shown in Figure 3. The horizontal axis corresponds to search time (milliseconds), and the vertical axis corresponds to search accuracy. From the results over the SIFT dataset shown in the first row of Figure 3, our approach performs the best. We can see that, given the target accuracy 90% 1-NN and 10-NN, our approach takes about $\frac{2}{3}$ time of the second best algorithm, iterative neighborhood graph search.

The second row of Figure 3 shows the results over the GIST dataset. Compared with the SIFT feature (a 128-dimensional vector), the dimension of the GIST feature (384) is larger and the search is hence more challenging. It can be observed that our approach is still consistently better than other approaches. In particular, the improvement is more significant, and for the target precision 70% our approach takes only half time of the second best approach, from 1 to 100 NNs. The third row of Figure 3 shows the results over the HOG dataset. This data set is the most difficult because it contains more $(10M)$ descriptors and its dimension is the largest (512). Again, our approach achieves the best results. For the target accuracy 70%, the search time in the case of 1 NN is about $\frac{4}{7}$ of the time of the second best algorithm.

All the neighborhood graph search algorithms outperform the other algorithms, which shows that the neighborhood graph structure is good to index vectors. The superiority of our approach to previous neighborhood graph algorithms stems from that our approach exploits the bridge graph to help the search. Inverted multi-index does not produce competitive results because its advantage is small index structure size but its search performance is limited by an unfavorable trade-off between the search accuracy and the time overhead in quantization. It is shown in [4] that inverted multi-index works the best when using a second-order multi-index and a large codebook, but this results in high quantization cost. In contrast, our approach benefits from the neighborhood graph structure so that we can use a high-order product quantizer to save the quantization cost.

In addition, we also conduct experiments to compare the source coding based ANN search algorithm [18]. This algorithm compresses each data vector into a short code using product quantization, resulting in the fast approximate distance computation between vectors. We report the results from the IVFADC system that performs the best as pointed in [18] over the 1M SIFT and GIST features. To compare IVFADC with our approach, we follow the scheme in [18] to add a verification stage to the IVFADC system. We cluster the data points into K inverted lists and use a 64-bits code to represent each vector as done in [18]. Given a query, we first find its M nearest inverted lists, then compute the approximate distance from the query to each of the candidates in the retrieved inverted lists. Finally we re-rank the top L candidates using Euclidean distance and compute the 1-recall [18] of the nearest neighbor (the same to the definition of the search accuracy for 1-NN). Experimental results show that $K = 2048$ gets superior performance. Figure 4 shows the results with respect to the parameters M and L. One can see that our approach gets superior performance.

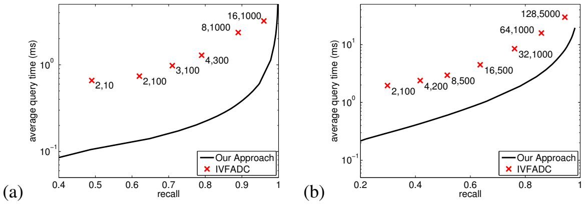


Figure 4. Search performances comparison with IVFADC [18] over (a) $1M$ SIFT and (b) $1M$ GIST. The parameters M (the number of inverted lists visited), L (the number of candidates for re-ranking) are given beside each marker of IVFADC.

Experiments over the BIGANN dataset. We evaluate the performance of our approach when combining it with the IVFADC system [18] for searching very large scale datasets. The IVFADC system organizes the data using inverted indices built via a coarse quantizer and represents each vector by a short code produced by product quantization. During the search stage, the system visits the inverted lists in ascending order of the distances to the query and re-ranks the candidates according to the short codes. The original implementation only uses a small number of inverted lists to avoid the expensive time cost in finding the exact nearest inverted indices. The inverted multi-index [4] is used to replace the inverted indices in the IVFADC system, which is shown better than the original IVFADC implementation [18].

We propose to replace the nearest inverted list identification using our approach. The good search quality of our approach in terms of both accuracy and efficiency makes it feasible to handle a large number of inverted lists. We quantize the $1B$ features into millions ($6M$ in our implementation) of groups using a fast approximate k-means clustering algorithm [40], and compute the centers of all the groups forming the vocabulary. Then we use our approach to assign each vector to the inverted list corresponding to the nearest center, producing the inverted indices. The residual displacement between each vector and its center is quan-

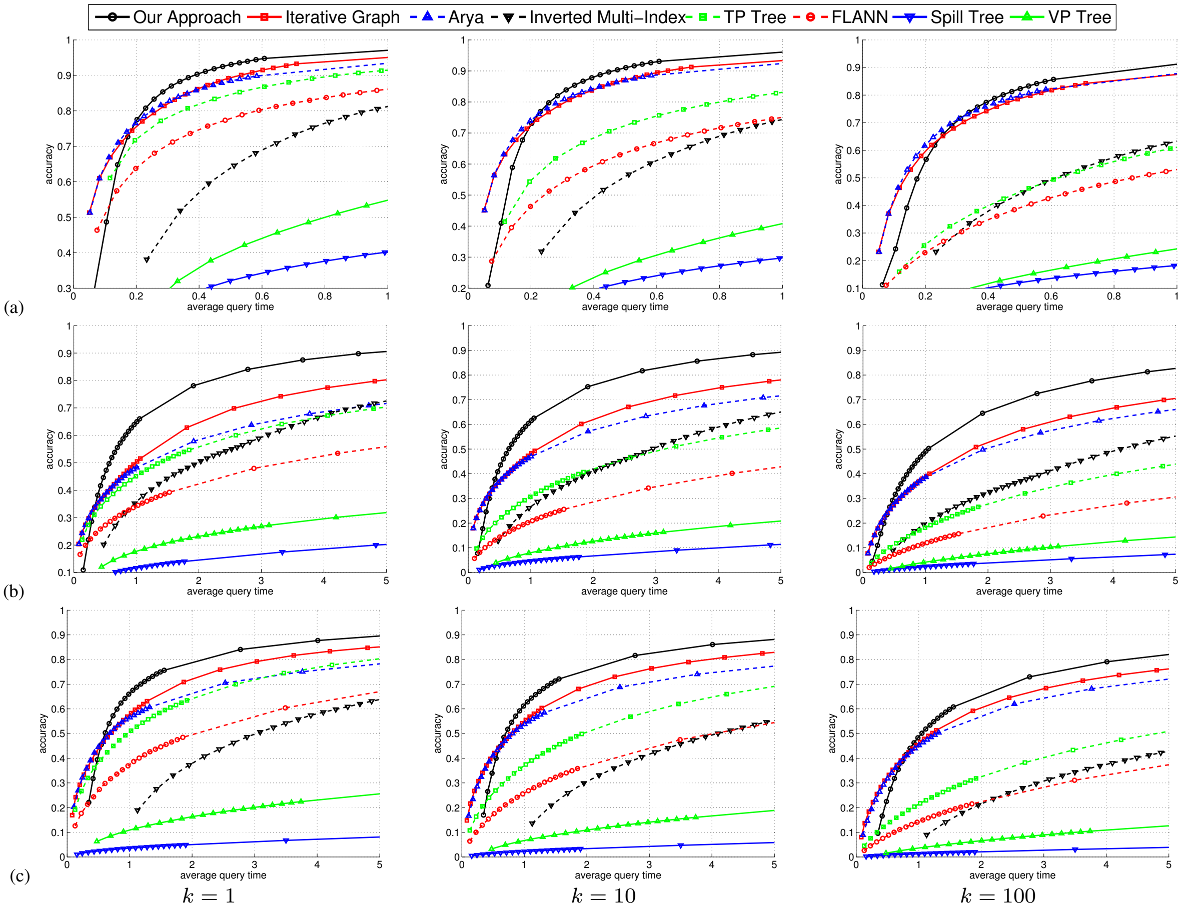


Figure 3. Performance comparison on (a) $1M$ 128-dimensional SIFT features, (b) $1M$ 384-dimensional GIST features, and (c) $10M$ 512-dimensional HOG features. k is the number of target nearest neighbors.

tized using product quantization to obtain extra bytes for re-ranking. During the search stage, we find the nearest inverted lists to the query using our approach and then do the same reranking procedure as in [4, 18]

Following [4, 18] we calculate the recall@T scores of the nearest neighbor with respect to different length of the visited candidate list L and different numbers of extra bytes, $m = 8, 16$. The recall@T score is equivalent to the accuracy for the nearest neighbor if a short list of T vectors is verified using exact Euclidean distances [19]. The performance is summarized in Table 2. It can be seen that our approach consistently outperforms Multi-D-ADC [4] and IVFADC [18] in terms of both recall and time cost when retrieving the same number of visited candidates. The superiority over IVFADC stems from that our approach significantly increases the number of inverted indices and produces space partitions with smaller (coarse) quantization errors and that our system accesses a few coarse centers while guarantees relatively accurate inverted lists. For inverted multi-index approach, although the total number of centers is quite large the data vectors are not evenly divided into inverted lists. As reported in the supplementary material of [4], 61% of the inverted lists are empty. Thus the quantization quality is not as good as ours. Consequently, it performs worse than our approach.

4. Analysis and discussion

Index structure size. In addition to the neighborhood graph and the reference vectors, the index structure of our approach includes a bridge graph and the bridge vectors. The number of bridge vectors in our implementation is $O(N)$, with N being the number of the reference vectors. The storage cost of the bridge vectors are then $O(\sqrt[m]{N})$, and the cost of the bridge graph is also $O(N)$. In the case of $1M$ 384-dimensional GIST byte-valued features, without optimization, the storage complexity ($125M$ bytes) of the bridge graph is smaller than the reference vectors ($384M$ bytes) and the neighborhood graph ($160M$ bytes). The cost of KD trees, VP trees, and TP trees are $\sim 180M$, $\sim 180M$, and $\sim 560M$ bytes. In summary, the storage cost of our index structure is comparable with those neighborhood graph and tree-based structures.

In comparison to source coding [18, 19] and hashing without using the original features, and inverted indices

Table 2. The performance (recall for the top-1, top-10, and top-100 candidates after reranking and average search time in milliseconds) comparison between IVFADC [19], Multi-D-ADC [4] and Our approach (Graph-D-ADC). IVFADC uses inverted lists with $K = 1024$, Multi-D-ADC uses the second-order multi-index with $K = 2^{14}$ and our approach use inverted lists with $K = 6M$

System	List len.	R@1	R@10	R@100	Time
BIGANN, 1 billion SIFTs, 8 bytes per vector					
IVFADC	4 million	0.100	0.280	0.600	960
Multi-D-ADC	10000	0.165	0.492	0.726	29
Multi-D-ADC	30000	0.172	0.526	0.824	44
Multi-D-ADC	100000	0.173	0.536	0.870	98
Graph-D-ADC	10000	0.199	0.562	0.802	24
Graph-D-ADC	30000	0.201	0.584	0.873	39
Graph-D-ADC	100000	0.201	0.589	0.896	90
BIGANN, 1 billion SIFTs, 16 bytes per vector					
IVFADC	4 million	0.220	0.610	0.890	1135
Multi-D-ADC	10000	0.324	0.685	0.755	30
Multi-D-ADC	30000	0.347	0.777	0.891	47
Multi-D-ADC	100000	0.354	0.813	0.959	109
Graph-D-ADC	10000	0.374	0.764	0.831	24
Graph-D-ADC	30000	0.391	0.829	0.924	39
Graph-D-ADC	100000	0.395	0.851	0.964	92

(e.g. [4]), our approach takes more storage cost. However, the search quality of our approach in terms of accuracy and time is much better, which leaves users for algorithm selection according to their preferences to less memory or less time. Moreover the storage costs for $1M$ GIST and SIFT features ($< 1G$ bytes) and even $10M$ HOG features ($< 8G$ bytes) are acceptable in most today's machines. When applying our approach to the BIGANN dataset of $1B$ SIFT features, the index structure size for our approach is about $14G$ for $m = 8$ and $22G$ for $m = 16$, which is similar with Multi-D-ADC [4] ($13G$ for $m = 8$ and $21G$ for $m = 16$) and IVFADC [18] ($12G$ for $m = 8$ and $20G$ for $m = 16$).

Construction complexity. The most time-consuming process in constructing the index structure in our approach is the construction of the neighborhood graph. Recent research [42] shows that an approximate neighborhood graph can be built in $O(N \log N)$ time, which is comparable to the cost of constructing the bridge graph. In our experiments, using a $3.4G$ Hz quad core Intel PC, the index structures of the $1M$ SIFT data, the $1M$ GIST data, and the $10M$ HOG data can be built within half an hour, an hour, and 10 hours, respectively. These time costs are relatively large but acceptable as they are offline processes.

The algorithm of combining our approach with the IVFADC system [18] over the BIGANN dataset of size 1 billion requires the similar construction cost with the state-of-the-art algorithm [4]. Because the number of data vectors is very large ($1B$), the most time-consuming stage is to assign each vector to the inverted lists and both take about 2 days. The structure of our approach organizing the $6M$ centers takes only a few hours, which is relatively small. These construction stages are all run with 48 threads on a server with 12 AMD Opteron $1.9G$Hz quad core processors.

Search complexity. The search procedure of our approach consists of the distance computation over the subvectors, the traversal over the bridge graph and the neighborhood graph. The distance computation over the subvectors is very cheap and takes small constant time (about the distance computation cost with 100 vectors in our experiments). Compared with the number of reference vectors that are required to reach an acceptable accuracy (e.g., the number is about 4800 for accuracy 90% in the $1M$ 384-dimensional GIST feature data set), such time cost is negligible.

Besides the computation of the distances between the query vector and the visited reference vectors, the additional time overhead comes from maintaining the priority queue and querying the bridge vectors using the multi-sequence algorithm. Given there are T reference vectors that have been discovered, it can be easily shown that the main queue is no longer than T. Consider the worst case that all the T reference vectors come from the bridge graph, where each bridge vector is associated with α unique reference vectors on average (the statistics for α in our experiments is presented in Table 1), we have that $\frac{T}{\alpha}$ bridge vectors are visited. Thus, the maintenance of the main queue takes $O((1 + \frac{1}{\alpha})T \log T)$ time. Extracting $\frac{T}{\alpha}$ bridge vectors using the multi-sequence algorithm [4] takes $O(\frac{T}{\alpha} \log(\frac{T}{\alpha}))$. Consequently the time overhead on average is $O((1 + \frac{2}{\alpha})T \log T - \frac{T}{\alpha} \log \alpha) = O(T \log T)$.

Figure 5 shows the time cost of visiting $10K$ reference vectors in different algorithms on two datasets. Linear scan represents the time cost of computing the distances between a query and all reference vectors. The overhead of a method is the difference between the time cost of this method and that of linear scan. We can see that the inverted multi-index takes the minimum overhead and our approach is the second minimum. This is because our approach includes extra operations over the main queue.

Relations to source coding [18] and inverted multi-index [4]. Product quantization (or generally Cartesian concatenation) has two attractive properties. One property is that it is able to produce a large set of concatenated vectors from several small sets of subvectors. The other property is that the exact nearest vectors to a query vector from such a large set of concatenated vectors can be quickly found using the multi-sequence algorithm. The application to source coding [18] exploits the first property, thus results in fast distance approximation. The application to inverted multi-index [4] makes use of the second property to fast retrieve concatenated quantizers. In contrast, our approach exploits both the two properties: the first property guarantees that the approximation error of the concatenated vectors to the reference vectors is small with small sets of subvectors, and the second property guarantees that the retrieval from the concatenated vectors is very efficient and hence the time overhead is small.

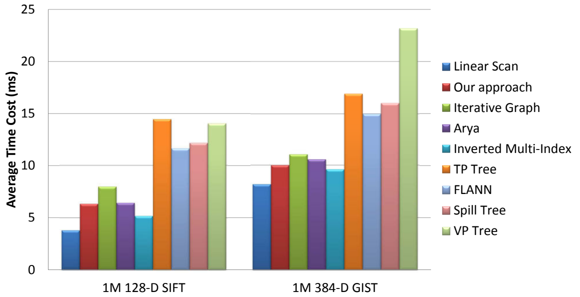


Figure 5. Average time cost of visiting $10K$ reference vectors. The time overhead (difference between the average time cost and the cost of liner scan) of our approach is comparably small.

5. Conclusions

The key factors contribute to the superior performance of our proposed approach include: (1) Discovering NN candidates from the neighborhood of both bridge vectors and reference vectors is very cheap; (2) The NN candidates from the neighborhood of the bridge vector have high probability to be true NNs because there are a large number of effective bridge vectors generated by Cartesian concatenation; (3) Retrieving nearest bridge vectors is very efficient. The algorithm is very simple and is easily implemented. The power of our algorithm is demonstrated by the superior ANN search performance over large scale SIFT, HOG, and GIST datasets, as well as over a very large scale dataset, the BIGANN dataset of 1 billion SIFT features through the combination of our approach with the IVFADC system.

Acknowledgements

The research work of Jing Wang, Gang Zeng and Rui Gan is supported by National Nature Science Foundation of China (NSFC Grant) 61375022, 61005037 and 90920304.

References

[1] K. Aoyama, K. Saito, H. Sawada, and N. Ueda. Fast approximate similarity search based on degree-reduced neighborhood graphs. In *KDD*, pages 1055–1063, 2011. 1, 2

[2] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *SODA*, pages 271–280, 1993. 1, 2, 4

[3] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45(6):891–923, 1998. 1

[4] A. Babenko and V. S. Lempitsky. The inverted multi-index. In *CVPR*, pages 3069–3076, 2012. 2, 3, 4, 5, 6, 7

[5] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR*, pages 1000–1006, 1997. 1

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. 1, 2

[7] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ICML*, pages 97–104, 2006. 1

[8] M. Brown and D. G. Lowe. Recognising panoramas. In *ICCV*, pages 1218–1227, 2003. 1

[9] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *STOC*, pages 537–546, 2008. 1

[10] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004. 1, 2

[11] L. Fei-Fei, R. Fergus, and P. Perona. Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories. In *CVPR 2004 Workshop on Generative-Model Based Vision*, 2004. 4

[12] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977. 1

[13] A. Frome, Y. Singer, F. Sha, and J. Malik. Learning globally-consistent local distance functions for shape-based image retrieval and classification. In *ICCV*, pages 1–8, 2007. 1

[14] Y. Gong and S. Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *CVPR*, pages 817–824, 2011. 1

[15] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI*, pages 1312–1317, 2011. 1

[16] J. Hays and A. A. Efros. Scene completion using millions of photographs. *ACM Trans. Graph.*, 26(3):4, 2007. 1

[17] Y. Hwang, B. Han, and H.-K. Ahn. A fast nearest neighbor search algorithm by nonlinear embedding. In *CVPR*, pages 3053–3060, 2012. 1

[18] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, 2011. 1, 2, 3, 5, 6, 7

[19] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*, pages 861–864, 2011. 2, 4, 6, 7

[20] Y. Jia, J. Wang, G. Zeng, H. Zha, and X.-S. Hua. Optimizing kd-trees for scalable visual descriptor indexing. In *CVPR*, pages 3392–3399, 2010. 1, 4

[21] B. Kulis and T. Darrells. Learning to hash with binary reconstructive embeddings. In *NIPS*, pages 577–584, 2009. 1

[22] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, 2009. 1

[23] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum. Real-time texture synthesis by patch-based sampling. *ACM Trans. Graph.*, 20(3):127–150, 2001. 1

[24] T. Liu, A. W. Moore, A. G. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, 2004. 1, 4

[25] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007. 1

[26] A. W. Moore. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *UAI*, pages 397–405, 2000. 1

[27] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISSAPP (1)*, pages 331–340, 2009. 1, 4

[28] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *CVPR (2)*, pages 2161–2168, 2006. 1, 2

[29] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *CVPR*, 2007. 1

[30] M. Raginsky and S. Lazebnik. Locality sensitive binary codes from shift-invariant kernels. In *NIPS*, 2009. 1

[31] H. Samet. *Foundations of multidimensional and metric data structures*. Elsevier, Amsterdam, 2006. 1, 2

[32] T. B. Sebastian and B. B. Kimia. Metric-based shape retrieval in large databases. In *ICPR (3)*, pages 291–296, 2002. 1, 2

[33] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*, 2008. 1

[34] J. Sivic and A. Zisserman. Efficient visual search of videos cast as text retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(4):591–606, 2009. 2

[35] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: exploring photo collections in 3D. *ACM Trans. Graph.*, 25(3):835–846, 2006. 1

[36] A. B. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(11):1958–1970, 2008. 2, 4

[37] W. Tu, R. Pan, and J. Wang. Similar image search with a tiny bag-of-delegates representation. In *ACM Multimedia*, pages 885–888, 2012. 2

[38] J. Wang and S. Li. Query-driven iterated neighborhood graph search for large scale indexing. In *ACM Multimedia*, pages 179–188, 2012. 1, 2, 4

[39] J. Wang, J. Wang, X.-S. Hua, and S. Li. Scalable similar image search by joint indices. In *ACM Multimedia*, pages 1325–1326, 2012. 2

[40] J. Wang, J. Wang, Q. Ke, G. Zeng, and S. Li. Fast approximate k-means via cluster closures. In *CVPR*, pages 3037–3044, 2012. 5

[41] J. Wang, J. Wang, N. Yu, and S. Li. Order preserving hashing for approximate nearest neighbor search. In *ACM Multimedia*, 2013. 1

[42] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable k-nn graph construction for visual descriptors. In *CVPR*, pages 1106–1113, 2012. 4, 7

[43] J. Wang, N. Wang, Y. Jia, J. Li, G. Zeng, H. Zha, and X.-S. Hua. Trinary-projection trees for approximate nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2013. 1, 4

[44] Y. Weiss, A. B. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, pages 1753–1760, 2008. 1

[45] H. Xu, J. Wang, Z. Li, G. Zeng, S. Li, and N. Yu. Complementary hashing for approximate nearest neighbor search. In *ICCV*, pages 1631–1638, 2011. 1

[46] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, pages 311–321, 1993. 1, 4