Approximate Nearest Neighbor Search Using Query-Directed Dense Graph

Hongya Wang[1,2](✉), Zeng Zhao[1], Kaixiang Yang[1], Hui Song[1], and Yingyuan Xiao[3]

[1] Donghua University, Shanghai, China
hywang@dhu.edu.cn

[2] Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai, China

[3] Tianjin University of Technology, Tianjin, China

Abstract. High-dimensional approximate nearest neighbor search (ANNS) has drawn much attention over decades due to its importance in machine learning and massive data processing. Recently, the graph-based ANNS become more and more popular thanks to the outstanding search performance. While various graph-based methods use different graph construction strategies, the widely-accepted principle is to make the graph as sparse as possible to reduce the search cost. In this paper, we observed that the sparse graph incurs significant cost in the high recall regime (close or equal to 100%). To this end, we propose to judiciously control the minimum angle between neighbors of each point to create more dense graphs. To reduce the search cost, we perform K-means clustering for the neighbors of each point using cosine similarity and only evaluate neighbors whose centroids are close to the query in angular similarity, i.e., query-directed search. PQ-like method is adopted to optimize the space and time performance in evaluating the similarity of centroids and the query. Extensive experiments over a collection of real-life datasets are conducted and empirical results show that up to 2.2x speedup is achieved in the high recall regime.

Keywords: Nearest neighbor search · Graph-based method · Query-directed search

1 Introduction

Nearest neighbor search (NNS) has been a hot topic over decades, which plays an important role in many applications such as data mining, machine learning and massive data processing. For high-dimensional NNS, due to the difficulty of finding exact results [8,9], most people turn to the approximate version of NNS, named Approximate Nearest Neighbor Search (ANNS). Recently, graph-based methods have gained much attention in answering ANNS. Given a finite point set S in $\mathbb{R}^D$, a graph is a structure composed of a set of nodes (representing

C. S. Jensen et al. (Eds.): DASFAA 2021 Workshops, LNCS 12680, pp. 429–444, 2021.
https://doi.org/10.1007/978-3-030-73216-5_29

a point in the dataset) and edges. If there is a neighbor relationship between two nodes, an edge is added between the two nodes. If each node links K edges, the graph is a KNN graph. The way to construct the graph affects greatly the search efficiency and precision, so many researchers are committed to improving the performance using different heuristics in the construction of the graph.

The common wisdom in constructing an KNN graph is to reduce the average out-degree as much as possible because the search cost is determined by the number of hops during walking the graph times the average out-degree. By graph theory, average out-degree and the connectivity of graphs are conflicting design goals. Hence, low average out-degree will make the graph too sparse and thus increase the difficulty of finding high quality kNN.

In this paper, we argue that one could obtain low search cost and high answer quality at one shot with affordable extra memory. Our first observation is that the state-of-the-art algorithms such as HNSW and NSG cannot achieve this goal even given enough extra memory, which will be discussed in details in Sect. 2. To tackle this problem, we propose to (1) control the minimum angle between neighbors of each point judiciously to create dense KNN graphs and thus improve the connectivity, and (2) use the query to guide the evaluation of neighbors of the base point, which significantly reduce the search cost. Figure 1 illustrates our idea using a simple example. In Fig. 1(a), the search algorithm examines all neighbors of o_1 and chooses the nearest one to q as the next base point. In contrast, o_2 has more neighbors than o_1 because the minimum angle between them is smaller. Suppose we are aware of the direction of q then we can only compare o_3 and o_4 with q, which reduces the number of distance evaluation dramatically. Note that, for almost all search graph construction, the memory used to store the graph depends on the maximum out-degree (MOD) instead the average out-degree for implementation efficiency. Thus, our method do not increase the memory cost to store the graph itself as will be discussed in Sect. 2.

Knowing the direction of q requires extra information associated with the graph. Our proposal is to partition neighbors of each point into clusters using standard clustering algorithms such as K-means. One modification is that we use *cosine similarity*, instead of the Euclidean distance, as the similarity measure. By comparing the cosine similarity between centroids and q we can avoid accessing distant neighbors and reduce the overall cost. The memory cost is very high (several times as much as the dataset) if we store the original centoids directly. Fortunately, slightly imprecise direction information is acceptable, which enables us to compress centroids using the product quantization method [18]. For example, an original centroid of dimension 128 needs 512 bytes to store and the compressed code occupies only 8 to 16 bytes.

To sum up, the main contributions of this paper are:

– We propose a novel query-directed dense graph (QDG) indexing method by controlling the minimum angle between neighbors and using clustering centroids to guide efficient search procedure. Please note that the design principles of QDG is orthogonal to specific graph construction algorithms, and thus are applicable for almost all graph-based methods.

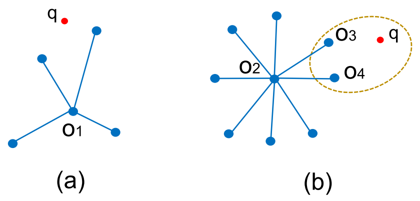


Fig. 1. An example to illustrate the main idea

- To improve space efficiency, we use modified product quantization (PQ) method to optimize the algorithm performance and reduce the index size.
- Extensive experiments show that QDG outperforms HNSW [24] and NSG [13], the two state-of-the-art graph algorithms in efficiency over a collection of real datasets. Particularly, up to 2.2x speedup is achieved at high recall regime.

This paper is organized as follows. Section 2 motivates our proposal. The details of QDG is presented in Sect. 3. Experimental results and analysis is given in Sect. 4. Related work is discussed in Sect. 5. Section 6 concludes this article.

2 Motivation

In recent graph-based methods, due to the high computational cost of building an exact KNN graph, many researchers turn to build an approximated KNN graph. Many experimental results such as Efanna [12] proved that the approximate KNN graph still performs well.

For almost all graph-based methods, the ANN search procedure is based on the same principle as follows. For a query q, start at an initial vertex chosen arbitrarily or using some sophisticated selection rule. Moves along an edge to the adjacent vertex with minimum distance to q. Repeat this step until the current element v is closer to q than all its neighbors, and then report v as the NN of q. Figure 4(a) illustrates the searching procedure for q in a sample graph starting from p.

In order to reduce the searching time on the graph, constructing an approximate KNN graph usually tends to reduce the out-degree of the graph. Out-degree refers to the number of neighbors connected to each node on the graph. For example, HNSW adopts the RNG's edge selection strategy to select the neighbors. It can reduce the out-degree to a constant $C_D + o(1)$, which is only related to the dimension D [24]. However, this edge selection strategy is too strict to provide sufficient edges. NSG adopts the MRNG's edge selection strategy, which is based on a directed graph. It can better ensure that each node on the graph has sufficient neighbors than RNG, and the angle between any two edges sharing the same node is at least $60°$ [13].

Through a number of preliminary experiments we observed that such edge selection policies may lead to too sparse graphs, especially for datasets such as Trevi and Nuswide. Table 1 lists the MOD and AOD for HNSW, NSG and the proposed method QDG, where AOD denotes the average out-degree over all points in the graph. By graph theory we know that the connectivity of graph is closed related to its AOD and if the graph is too sparse, the traversal length of a query will increase, which in turn decreases the efficiency [25].

Table 1. Comparison of the out-degree of graph in three methods. HNSW contains multiple graphs and we only report the AOD and MOD of its bottom-layer graph (HNSW0) here.

Dataset	$HNSW_0$		NSG		QDG	
	MOD	AOD	MOD	AOD	MOD	AOD
Audio	70	13	70	17	70	65
Sun	70	13	70	24	70	66
Cifar	70	17	70	29	70	69
Nuswide	70	4	70	8	70	16
Trevi	70	5	70	8	70	46

However, simply increasing MOD does not make out-degree greater because the edge selection policies such as RNG and MRNG set the lower bound of the minimum angle between any two edges sharing the same node. Table 2 lists the recall and search time at high recall regime for four datasets using NSG[1], which suggests (1) increasing MOD does not change the average out-degree much, and (2) adding more memory to the index cannot trade space to speed by using the existing index structure alone. Please note the index size is determined by MOD, instead of AOD, for almost all existing graph-based algorithms for implementation efficiency.

Table 2. Comparison of recall and cost on different datasets by increasing MOD.

MOD	Audio		Sun		Nuswide		Trevi	
	Recall	Cost	Recall	Cost	Recall	Cost	Recall	Cost
70	0.999	0.4403	0.9997	0.07573	0.7685	0.01964	0.9920	0.03730
100	0.999	0.4403	0.9997	0.07774	0.7685	0.02037	0.9935	0.03850
160	0.999	0.4403	0.9997	0.07820	0.7700	0.02079	0.9942	0.03962
220	0.999	0.4403	0.9997	0.07827	0.7702	0.02085	0.9952	0.03999
500	0.999	0.4403	0.9997	0.07827	0.7700	0.02086	0.9952	0.04002

[1] HNSW exhibits similar trends.

These two observations motivate us to increase the out-degree to ensure that there are sufficient neighbors for each point, that is, improving the connectivity. The side-effect of dense graph is the increasing computational cost because all neighbors of each point along the search path will have to be examined. To solve this problem, we give higher priority to the neighbors closer to the query, which will be discussed in next section.

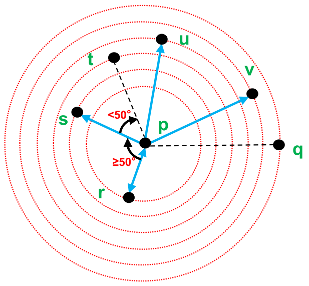


Fig. 2. The neighbor selection strategy at point p. All candidate neighbors are sorted by distance to p. Starting from the nearest point r, the neighbors whose angle is larger than the specified degree (e.g. 50°) will be reserved. The dotted line represents the abandoned neighbor, the directed arrows point to the reserved neighbor.

3 Query-Directed Dense Graph Algorithm

3.1 Graph Construction

QDG consists of three stages in search graph construction.

The first stage is to construct an approximate KNN graph. We use the same method as NSG in this stage [13]. After constructing the approximate KNN graph, the approximate center of the dataset will be calculated, which is called the Navigating Node. When we choose neighbor candidate sets for a point p, it will be treated as a query, and the greedy-search algorithm will be performed starting from the navigating node on the approximate KNN graph. During the search, the candidate set will be sorted by the distance to p and used for neighbor selection in the second stage.

Instead of using MRNG's edge selection adopted by NSG, we adjust the number of neighbors for each point by controlling the minimum angle between its neighbors. The edge selection strategy in the second stage is shown in Fig. 2. Assume that the minimum angle is 50°. First, the point r closest to p is selected and put it into the result set. When selecting the remaining edges, it will be selected from the candidate set according to the distance ranking with respect to point p. If the angle between itself and the existing ones is greater than 50°, it will be kept (like s in Fig. 2) and discarded otherwise (like t Fig. 2). The choice of minimum angle directly affects the average out-degree of the graph and is left for user to determine according to the dataset property.

The third stage is illustrated in Fig. 3. Each point on the graph has a set of neighbors, then we use K-means algorithm to cluster neighbors that are close to each other in angular similarity. Since the standard K-means algorithm only support the Euclidean distance, we make the following pre-processing. As we all know, the Euclidean distance between point A and point B in high-dimensional space is calculated as follows:

$$\|A-B\|^2 = (A-B)^T(A-B) = \|A\|^2 + \|B\|^2 - 2\,A^T B.$$

If A and B are normalized to unit vectors, i.e., $\|A\|^2 = \|B\|^2 = 1$, then $\|A-B\|^2$ is equal to $2(1 - cos(A,B))$, which means there is a monotonic relationship between the Euclidean distance and cosine similarity. As shown in Fig. 3, we first transform all candidate neighbors of point p into unit vectors w.r.t. p, then we use the K-means algorithm to cluster all unit vectors by the Euclidean distance (cosine similarity). The number of cluster centers $\mathcal{K}$ is specified by users and in Fig. 3 $\mathcal{K} = 4$.

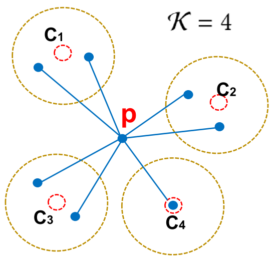


Fig. 3. The blue points are neighbors of point p on the graph, and the number of cluster centroids is 4. (Color figure online)

3.2 *k*NN Search on QDG

Most graph-based search algorithms use the greedy-search algorithm to identify kNN of a query. The only difference between the general search method and QDG is that we focus on reducing the out-degree at search stage, instead of the index construction stage. Figure 4(a) and Fig. 4(b) depict examples of the general greedy-search algorithm and QDG's search strategy, respectively. As shown in Fig. 4(a), the general greedy-search algorithm initializes the dynamic candidate set as the starting point p and its neighbors first. In the candidate set, the point closest to the query point is selected as the new starting point for the next iteration and visited points will be marked. The candidate set is of fixed size, which is often greater than k, and points in the candidate set are sorted according to the distance from the query point. This method can quickly reach the neighborhood of the query point. When all the points in the candidate set are examined, the iteration ends, and the algorithm returns the first k points in candidate set as kNN.

The search procedure of QDG differs from the general one mainly in the neighbor selection policy. Particularly, we specify the number of clusters k' to be checked during the search. As shown in Fig. 4(b), the number of clusters $\mathcal{K}$ is 3. Point 1, point 2 and point 3, 4 are in three different clusters, respectively. When starting from p, we calculate the cosine similarity between three cluster centroids and q. If we specify $k' = 1$, then we only need to check point 3 and point 4, which reduces the search cost significantly. k' and $\mathcal{K}$ are two tuning knobs determined by users.

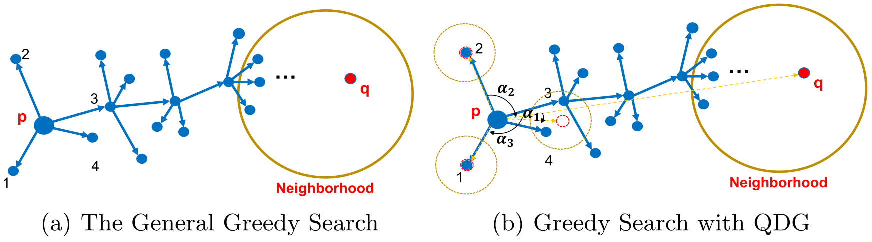


Fig. 4. (a) An example of the general search procedure. (b) The greedy-search algorithm of QDG. Point p is the entry node, point q is the query node, and the dark yellow circle represents the kNN neighborhood of q. The red dashed circle represents the cluster centroid of p and α_1, α_2, α_3 represent the angles between the cluster centroids and q, respectively. (Color figure online)

3.3 Space and Time Performance Optimization

Suppose that the dataset consists of n points and the number of clusters is $\mathcal{K}$, additional space for storing $n * \mathcal{K}$ vectors is required, which is unacceptable. Suppose, for any point, the number of cluster centroids of its neighbors is the close to the number of its neighbors, it will become meaningless to do clustering. To solve this problem, we only cluster the points with more than L neighbors, where L is set to 10 by default in this paper.

For points of which the number of their neighbors are greater than L, we use PQ to compress the centroid vectors. Specifically, the cluster centroids are used to train a codebook C and all the original centroid vectors are stored in compressed code form, which will greatly reduce the index storage cost. Figure 5 depicts a simple codebook trained using PQ, where the number of subvectors $m = 4$ and the number of centroids $k^* = 4$. Using this codebook, a vector of 16*4=64 bytes could be compressed into a one-byte code[2]. Please refer to Sect. 5.3 and [18] for more details about PQ.

Besides evaluating the Euclidian distance between candidate points and the query, the most time-consuming part of this algorithm is to calculate the cosine similarity between each cluster centroid and the query vector. To reduce such computation cost, we adopt a pre-calculation method similar to PQ.

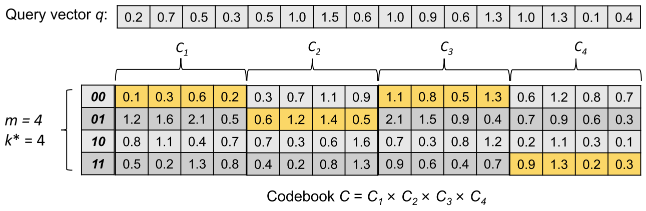
At online search stage, suppose a cluster centroid p of code 00010011 to be evaluated, the formula for calculating the cosine similarity between q and p is

[2] The dimension of the vector is 16 and it takes four bytes to store a float number.

as follows, where the dimension of p and q is 16 as illustrated in Fig. 5. The re-constructed vector of p from its code is the elements with yellow background color.

$$\cos(\boldsymbol{p},\boldsymbol{q}) = \frac{\boldsymbol{p}\cdot\boldsymbol{q}}{\|\boldsymbol{p}\|\cdot\|\boldsymbol{q}\|}$$

Since the length of $\boldsymbol{q}$ does not affect the ranking of the cosine similarity of different cluster centers, we do not computer $\|\boldsymbol{q}\|$. $\|\boldsymbol{p}\|$ can be obtained through the pre-calculation table constructed at indexing stage, which is illustrated in Fig. 6(a) (the root of sum of elements with yellow background color). Each element in this table is computed as the square sum of corresponding elements in the codebook. For example, the first element 0.50 in row one in Fig. 6(a) is equal to the square sum of the first four elements in the first row in Fig. 5. Similarly, the inner product pre-calculation table is illustrated in Fig. 6(b). By looking up the pre-calculation tables, the cosine similarity between p and q can be approximately computed by looking up these tables as $\cos(\boldsymbol{p},\boldsymbol{q}) = \frac{11.03}{3.30\times3.38} = 0.98$ since $\boldsymbol{p}\cdot\boldsymbol{q} = 0.59+3.90+3.81+2.73 = 11.03$, $\|\boldsymbol{p}\| = \sqrt{0.50+4.01+3.79+2.63} = 3.30$ and $\|\boldsymbol{q}\| = 3.38$. The ranking of cosine similarity of all cluster centroids can be obtained with these approximations.



Query vector q:

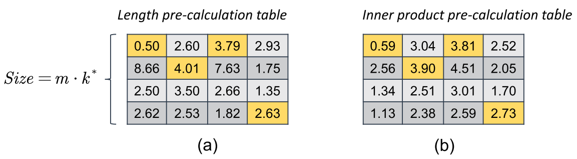
0.2	0.7	0.5	0.3	0.5	1.0	1.5	0.6	1.0	0.9	0.6	1.3	1.0	1.3	0.1	0.4

$m = 4$, $k^* = 4$

	C_1				C_2				C_3				C_4			
00	0.1	0.3	0.6	0.2	0.3	0.7	1.1	0.9	1.1	0.8	0.5	1.3	0.6	1.2	0.8	0.7
01	1.2	1.6	2.1	0.5	0.6	1.2	1.4	0.5	2.1	1.5	0.9	0.4	0.7	0.9	0.6	0.3
10	0.8	1.1	0.4	0.7	0.7	0.3	0.6	1.6	0.7	0.3	0.8	1.2	0.2	1.1	0.3	0.1
11	0.5	0.2	1.3	0.8	0.4	0.2	0.8	1.3	0.9	0.6	0.4	0.7	0.9	1.3	0.2	0.3

Codebook $C = C_1 \times C_2 \times C_3 \times C_4$

Fig. 5. Query vector and codebook. (best viewed in color)



$Size = m \cdot k^*$

Length pre-calculation table

0.50	2.60	3.79	2.93
8.66	4.01	7.63	1.75
2.50	3.50	2.66	1.35
2.62	2.53	1.82	2.63

(a)

Inner product pre-calculation table

0.59	3.04	3.81	2.52
2.56	3.90	4.51	2.05
1.34	2.51	3.01	1.70
1.13	2.38	2.59	2.73

(b)

Fig. 6. (a) Length pre-calculation table and (b) inner product pre-calculation table. (best viewed in color)

4 Experiments

In this section, we conduct a detailed analysis using publicly available datasets to show the efficiency of QDG. The design principles of QDG are orthogonal to specific graph-based search methods. In this paper, we only report the results using NSG. We first describe the datasets and the parameters used, and then we present the results and analysis.

4.1 Datasets and Experiment Setting

Our experiment uses five datasets, Audio, Sun, Cifar, Nuswide and Trevi. All the datasets we used can be found on Github[3]. The detailed information on the datasets is listed in Table 3. A set of 200 queries are randomly chosen from each dataset and then removed from the original dataset. We carried out comprehensive experiments with different k and the results exhibit similar trends. Due to space limitation, we only report the results for top-100 queries.

The MOD are all set to 70 for all three methods and other important parameter settings are listed in Table 3 as well. As we can see from Table 1, QDG graph are far more dense than HNSW and NSG because we decrease the minimum angle between neighbors of points. The number of clusters $\mathcal{K}$ in graph construction and the number of cluster centroid examined during the NN search k' are tuned to be the optimal. For cluster centroid compression, each vector is partitioned into $m = 8$ subvectors and k^* is set to 256. This incurs 8 bytes per cluster centriod extra memory for indexing. Please note the original index space cost for all three methods are determined by the MOD, which are all equal to $70 * 4 = 280$ bytes.

Table 3. Statistics of datasets and parameter settings.

Dataset	Dimension	No. of points	No. of queries	Minimum Angle	$\mathcal{K}$	k'	m	k^*
Audio	192	53,387	200	50°	9	7	8	256
Sun	512	79,106	200	50°	9	7	8	256
Cifar	512	50,000	200	40°	8	6	8	256
Nuswide	500	268,643	200	45°	9	7	8	256
Trevi	4096	99,900	200	50°	8	7	8	256

4.2 Evaluation Measures

In order to measure the performance of different algorithms, we use the *average recall* as a criterion for evaluating accuracy. Given a query point, all the algorithms are expected to return k points. We need to compare how many of these k points are in the true k nearest neighbors. Suppose the returned set of k points for a query is R' and the true k nearest neighbors set of the query is R, the recall is defined as:

$$\text{recall} = \frac{|R' \cap R|}{|R|}$$

The *average recall* is the average over all the query points.

Another performance measure is the *average cost*. At online search stage, the number of Euclidean distance calculation with the query will be counted[4].

[3] https://github.com/DBWangGroupUNSW/nns_benchmark.

[4] For QDG, the number of evaluation of cluster centroids and the query is also counted.

Suppose the number is c and the total number of points in the dataset is n. Then the cost is defined as:

$$\text{cost} = \frac{c}{n}$$

The *average cost* is the average over all the query points. Usually, the smaller the average cost is, the shorter the search time will be.

4.3 Baseline Algorithms

The algorithms we choose to compare are the two state-of-the-art, i.e., NSG and HNSW. They are implemented in C++. We do not compare the non-graph methods because they have been shown less efficient by many researchers [13,20]. Since it is desirable to obtain high-precision results in real scenarios, we focus on the performance of all algorithms in the high-precision region.

There are many algorithms that do not support multi-threading at searching stage, so we use single thread setting to compare when searching. Most of these methods support multi-threading at indexing stage. To save time, we use eight threads when building the index.

HNSW is based on a hierarchical graph structure, which was proposed in [24]. In [22,23,27] authors have proposed a proximity graph k-ANNS algorithm called Navigable Small World (NSW). HNSW is an improved version of NSW and has a huge improvement in performance. HNSW has multiple implementation versions, such as Faiss, hnswlib[5]. We use hnswlib since it performs better than Faiss implementation.

NSG is a method based on KNN graph, in which the neighbor set of each point on this graph is pruned by the MRNG method. This method was first proposed in [13]. At search stage, each query point starts searching from the same navigating node. NSG can approximate MRNG very well and try to ensure a monotonic search path in the search procedure. Besides, NSG shows superior performance in the E-commercial search scenario of Taobao (Alibaba Group) and has been integrated into their search engine.

All methods, including QDG, are written in C++ and compiled by g++ 5.4 with "O3" option. The experiments on all datasets are carried out on a computer with i5-8300H CPU and 40 GB memory. Please note our design principles are also applicable for other graph-based search methods besides NSG.

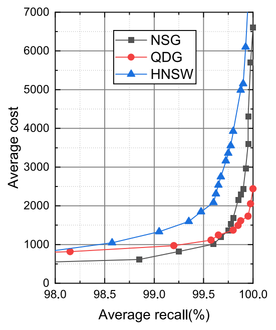
4.4 Results and Analysis

Recall Vs. Cost. The recall-cost curves of three algorithms on different datasets are shown in Fig. 7. From these figures we can see:

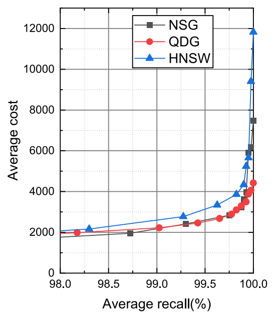
1. The cost of HNSW is constantly inferior to NSG and our method. This agrees with the result reported in [13]. Since QDG can be viewed as an enhanced version of NSG, it also performs better than HNSW on all five datasets.

[5] https://github.com/nmslib/hnswlib.

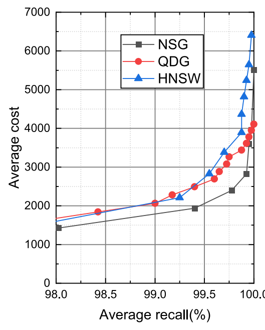
2. For Nuswide dataset, QDG beats NSG all the time. This can be explained by the fact that the AOD of QDG is two times as much as that of NSG (Table 1). Too sparse graph leads to weak connectivity, which results in long search path and high cost. In contrast, the dense graph of QDG provides much stronger connectivity and thus lower cost. Particularly, NSG examined 5276 points on average whereas QDG vistited 4047 points (centroids included) at recall 76.7, which translates to 30% performance gain.
3. For the remaining four datasets, QDG performs almost the same as or slightly worse than NSG in the relatively low recall region. The reason is that the connectivity of NSG already could provide fine accuracy at low cost and QDG are far more dense, which incurs slightly higher cost even with the help of query-directed pruning. However, the trend changes after a critical point in the high recall regime. Particularly, the recalls at the transition point are around 99.65%, 99.9%, 99.95% and 98% for Audio, Sun, Cifar and Trevi, respectively. After the critical point, the cost of NSG increase dramatically whereas QDG enjoys more smooth incline. For example, QDG achieves 2.7x, 1.7x and 1.34x speedup over NSG at recall of 100% for Audio, Sun and Cifar, respectively. For Trevi, the cost of NSG is 1.53 times as much as that of QDG at recall of 98.95%. The main reason is that QDG is dense enough to provide high recall while NSG has to search a way longer path to achieve the same recall.



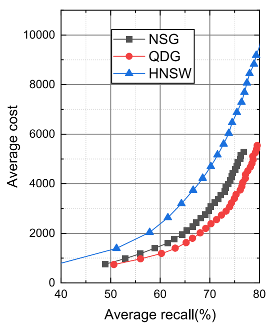
(a) Audio



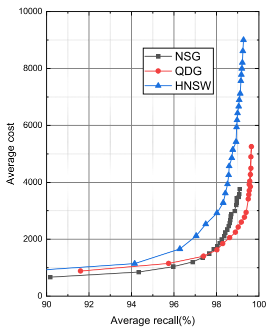
(b) Sun



(c) Cifar



(d) Nuswide



(e) Trevi

Fig. 7. The recall-cost curves of three algorithms on different datasets.

Recall Vs. Time. The time-recall curves of three algorithms on different datasets are shown in Fig. 8. Similar trends are observed as in Fig. 7 since the wall-clock search time are proportional to the cost. Particularly, QDG constantly outperforms NSG with around 10% performance gain on Nuswide and achieves 2.2x, 1.51x and 1.08x speedup over NSG at recall of 100% for Audio, Sun and Cifar, respectively. For Trevi, the cost of NSG is 1.29 times as much as that of QDG at recall of 98.95%. The speedup is slightly smaller than that in the case of Recall vs. Cost because it takes time to build the pre-calculation tables. More importantly, the accuracy of NSG saturates once reaching 99% whereas QDG achieve higher recall that the other algorithms cannot provide.

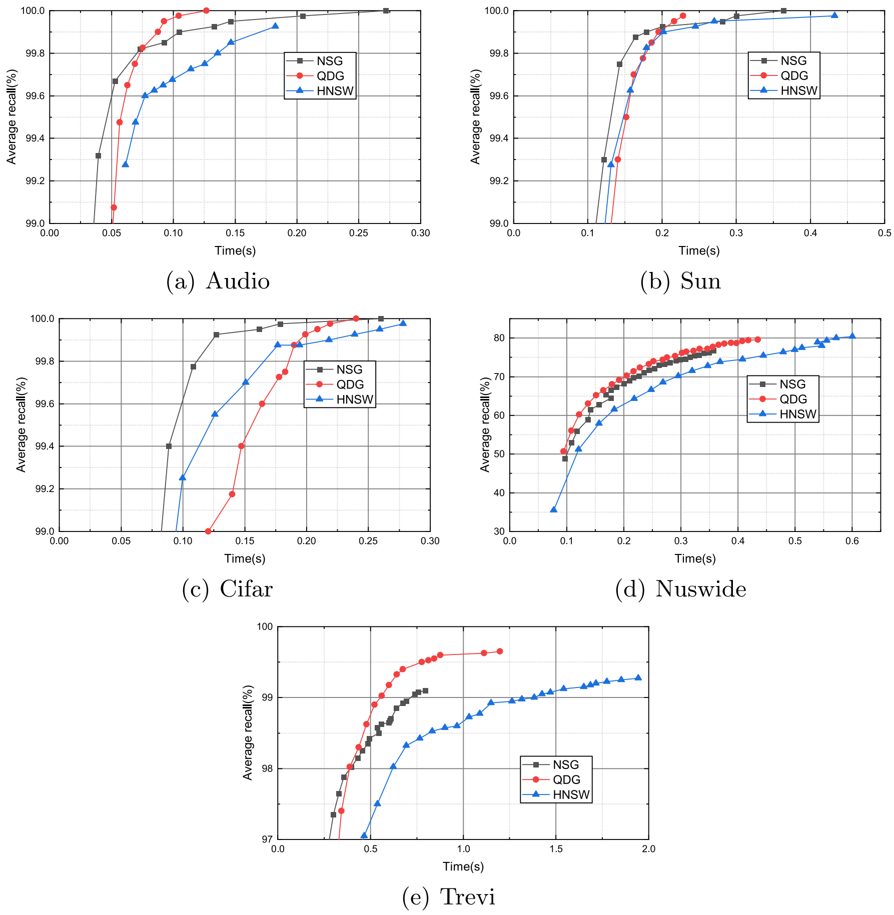


Fig. 8. The time-recall curves of all algorithms on different datasets.

5 Related Work

Approximate nearest neighbor search (ANNS) has been a hot topic over decades, it provides fundamental support for many applications of data mining, databases and information retrieval [2,11,29,31]. There is a large amount of significant literature on algorithms for approximate nearest neighbor search, which are mainly divided into the following categories: tree-structure based approaches, hashing-based approaches, quantization-based approaches, and graph-based approaches.

5.1 Tree-Structure Based Approaches

Hierarchical structures (tree) based methods offer a natural way to continuously partition a dataset into discrete regions at multiple scales, such as KD-tree [6,7], R-tree [10], SR-tree [19]. These methods perform very well when the dimensionality of the data is relatively low. However, it has been proved to be inefficient when the dimensionality of data is high. It has been shown in [30] that when the dimensionality exceeds about 10, existing indexing data structures based on space partitioning are slower than the brute-force, linear-scan approach. Many new hierarchical-structure-based methods [12,26] are presented to address this limitation.

5.2 Hashing-Based Approaches

For high-dimensional approximate search, the well-known indexing method is locality sensitive hashing (LSH) [15]. The main idea is to use a family of locality-sensitive hash functions to hash nearby data points into the same bucket. After the query point goes through the same hash functions, it will get the corresponding bucket number, and only compare the distance between the point in the bucket and the query point. In the end, the k approximate nearest neighbor results that are closest to the query point will be returned. In recent two decades, many LSH-based variants have been proposed, such as QALSH [17], Multi-Probe LSH [21], BayesLSH [28]. However, there is no guarantee that all the neighbor vectors will fall into the nearby buckets. In order to achieve a high recall (the number of true neighbors within the returned points set divides by the number of required neighbors), a large number of hash buckets need to be checked.

5.3 Quantization-Based Approaches

The most common of quantization-based methods is product quantization (PQ) [18]. It seeks to perform a similar dimension reduction to hashing algorithms, but in a way that better retains information about the relative distances between points in the original vector space. Formally, a quantizer is a function q mapping a D-dimensional vector $x \in \mathbb{R}^D$ to a vector $q(x) \in C = \{c_i; i \in \mathcal{I}\}$, where the index set $\mathcal{I}$ is from now on assumed to be finite: $\mathcal{I} = 0 \ldots k-1$. The reproduction values c_i are called centroids. The set $\mathcal{V}_i$ of vectors mapped to given index i is referred to as a cell, and defined as

$$\mathcal{V}_i \triangleq \left\{x \in \mathbb{R}^D : q(x) = c_i\right\}$$

The k cells of a quantizer form a partition of $\mathbb{R}^D$. So all the vectors lying in the same cell $\mathcal{V}_i$ are reconstructed by the same centroid c_i. Due to the huge number of samples required and the complexity of learning the quantizer, PQ uses m distinct quantizers to quantize the subvectors separately. An input vector will be divided into m distinct subvectors u_j, $1 \leq j \leq m$. The dimension of each subvector is $D^* = D/m$. An input vector x is mapped as follows:

$$\underbrace{x_1, \ldots, x_{D^*}}_{u_1(x)}, \cdots, \underbrace{x_{D-D^*+1}, \ldots, x_D}_{u_m(x)} \rightarrow q_1\left(u_1(x)\right), \ldots, q_m\left(u_m(x)\right)$$

where q_j is a low-complexity quantizer associated with the j^{th} subvector. And the codebook is defined as the Cartesian product,

$$\mathcal{C} = \mathcal{C}_1 \times \ldots \times \mathcal{C}_m$$

and a centroid of this set is the concatenation of centroids of the m subquantizers. All subquantizers have the same finite number k^* of reproduction values, the total number of centroids is $k = (k^*)^m$.

PQ offers three attractive properties: (1) PQ compresses an input vector into a short code (e.g., 64-bits), which enables it to handle typically one billion data points in memory; (2) the approximate distance between a raw vector and a compressed PQ code is computed efficiently (the so-called asymmetric distance computation (ADC) and the symmetric distance computation (SDC)), which is a good estimation of the original Euclidean distance; and (3) the data structure and coding algorithm are simple, which allow it to hybridize with other indexing structures. Because these methods avoid distance calculations on the original data vectors, it will cause a loss of certain calculation accuracy. When the recall rate is close to 1.0, the required length of the candidate list is close to the size of the dataset. Many quantization-based methods try to reduce quantization errors to improve calculation accuracy, such as SQ, Optimal Product Quantization (OPQ) [14], Tree Quantization (TQ) [3].

5.4 Graph-Based Approaches

Recently, graph-based methods have drawn considerable attention, such as NSG [13], HNSW [24], Efanna [12], and FANNG [16]. Graph-based methods construct a KNN graph offline, which can be regard as a big network graph in high-dimensional space [4,5]. However, the construction complexity of the exact KNN graph will increase exponentially. Hence, many researchers turn to building an approximated KNN graph.

Many graph-based methods perform well in search time, such as Efanna [12], KGraph [1], HNSW and NSG. They all use different neighbor selection methods to reduce the average out-degree. As we have shown in this paper, too sparse graph may jeopardize the performance at the high recall region.

6 Conclusion

In this paper, we proposed a new approximate nearest neighbor search method called QDG. This method is constructed based on the approximate KNN graph, and neighbors are selected according to the minimum angle between the neighbors of each point. To guide the search path using the query point, we cluster the neighbors of all points with cosine similarity in advance and only compare the clusters close to the query point in angular similarity at NN search stage. Extensive experiments indicates that our method perform better than the two state-of-the-art, NSG and HNSW, especially in the high recall regime.

Acknowledgments. The work reported in this paper is partially supported by NSFC under grant number (No: 61370205), NSF of Xinjiang Key Laboratory under grant number (No:2019D04024) and Tianjin "Project + Team" Key Training Project under Grant No. XC202022.

References

1. KGraph. https://github.com/aaalgo/kgraph
2. Arora, A., Sinha, S., Kumar, P., Bhattacharya, A.: Hd-index: Pushing the scalability-accuracy boundary for approximate knn search in high-dimensional spaces. arXiv preprint arXiv:1804.06829 (2018)
3. Babenko, A., Lempitsky, V.: Tree quantization for large-scale similarity search and classification. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4240–4248 (2015)
4. Baranchuk, D., Babenko, A.: Towards similarity graphs constructed by deep reinforcement learning. CoRR abs/1911.12122 (2019)
5. Baranchuk, D., Persiyanov, D., Sinitsin, A., Babenko, A.: Learning to route in similarity graphs. ICML **97**, 475–484 (2019)
6. Beis, J.S., Lowe, D.G.: Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In: Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 1000–1006. IEEE (1997)
7. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
8. Beyer, K., Goldstein, J., Ramakrishnan, R., Shaft, U.: When is "nearest neighbor" meaningful? In: Beeri, C., Buneman, P. (eds.) ICDT 1999. LNCS, vol. 1540, pp. 217–235. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49257-7_15
9. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. ACM Comput. Surv. (CSUR) **33**(3), 322–373 (2001)
10. Boston, M., et al.: A dynamic index structure for spatial searching. In: Proceedings of the ACM-SIGMOD, pp. 547–557 (1984)
11. Chen, L., Özsu, M.T., Oria, V.: Robust and fast similarity search for moving object trajectories. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 491–502 (2005)
12. Fu, C., Cai, D.: Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. arXiv preprint arXiv:1609.07228 (2016)
13. Fu, C., Xiang, C., Wang, C., Cai, D.: Fast approximate nearest neighbor search with the navigating spreading-out graph. arXiv preprint arXiv:1707.00143 (2017)

14. Ge, T., He, K., Ke, Q., Sun, J.: Optimized product quantization for approximate nearest neighbor search. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2946–2953 (2013)
15. Gionis, A., Indyk, P., Motwani, R., et al.: Similarity search in high dimensions via hashing. Vldb **99**, 518–529 (1999)
16. Harwood, B., Drummond, T.: Fanng: fast approximate nearest neighbour graphs. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 5713–5722 (2016)
17. Huang, Q., Feng, J., Zhang, Y., Fang, Q., Ng, W.: Query-aware locality-sensitive hashing for approximate nearest neighbor search. Proc. VLDB Endow. **9**(1), 1–12 (2015)
18. Jegou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. IEEE Trans. Pattern Anal. Mach. Intell. **33**(1), 117–128 (2010)
19. Katayama, N., Satoh, S.: The SR-tree: an index structure for high-dimensional nearest neighbor queries. ACM Sigmod Rec. **26**(2), 369–380 (1997)
20. Li, W., Zhang, Y., Sun, Y., Wang, W., Zhang, W., Lin, X.: Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement (v1.0). CoRR abs/1610.02455 (2016)
21. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe LSH: efficient indexing for high-dimensional similarity search. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 950–961 (2007)
22. Malkov, Y., Ponomarenko, A., Logvinov, A., Krylov, V.: Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces. In: Navarro, G., Pestov, V. (eds.) SISAP 2012. LNCS, vol. 7404, pp. 132–147. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32153-5_10
23. Malkov, Y., Ponomarenko, A., Logvinov, A., Krylov, V.: Approximate nearest neighbor algorithm based on navigable small world graphs. Inf. Syst. **45**, 61–68 (2014)
24. Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Trans. Pattern Anal. Mach. Intell. (2018)
25. Newman, M.: Networks: An Introduction. Oxford University Press (2010)
26. Nister, D., Stewenius, H.: Scalable recognition with a vocabulary tree. In: 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2006), vol. 2, pp. 2161–2168. IEEE (2006)
27. Ponomarenko, A., Malkov, Y., Logvinov, A., Krylov, V.: Approximate nearest neighbor search small world approach. In: International Conference on Information and Communication Technologies & Applications, vol. 17 (2011)
28. Satuluri, V., Parthasarathy, S.: Bayesian locality sensitive hashing for fast similarity search. arXiv preprint arXiv:1110.1328 (2011)
29. Teodoro, G., Valle, E., Mariano, N., Torres, R., Meira, W., Saltz, J.H.: Approximate similarity search for online multimedia services on distributed CPU-GPU platforms. VLDB J. **23**(3), 427–448 (2014)
30. Weber, R., Schek, H.J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. VLDB **98**, 194–205 (1998)
31. Zheng, Y., Guo, Q., Tung, A.K., Wu, S.: Lazylsh: approximate nearest neighbor search for multiple distance functions with a single index. In: Proceedings of the 2016 International Conference on Management of Data, pp. 2023–2037 (2016)