Pruned Bi-directed K-nearest Neighbor Graph for Proximity Search

Masajiro Iwasaki[(✉)]

Yahoo Japan Corporation, Tokyo, Japan
`miwasaki@yahoo-corp.jp`

Abstract. In this paper, we address the problems with fast proximity searches for high-dimensional data by using a graph as an index. Graph-based methods that use the k-nearest neighbor graph (KNNG) as an index perform better than tree-based and hash-based methods in terms of search precision and query time. To further improve the performance of the KNNG, the number of edges should be increased. However, increasing the number takes up more memory, while the rate of performance improvement gradually falls off. Here, we propose a pruned bi-directed KNNG (PBKNNG) in order to improve performance without increasing the number of edges. Different directed edges for existing edges between a pair of nodes are added to the KNNG, and excess edges are selectively pruned from each node. We show that the PBKNNG outperforms the KNNG for SIFT and GIST image descriptors. However, the drawback of the KNNG is that its construction cost is fatally expensive. As an alternative, we show that a graph can be derived from an approximate neighborhood graph, which costs much less to construct than a KNNG, in the same way as the PBKNNG and that it also outperforms a KNNG.

1 Introduction

How to conduct fast proximity searches of large-scale high dimensional data is an inevitable problem not only for similarity-based image retrieval and image recognition but also for multimedia data processing and large-scale data mining. Image descriptors, especially local descriptors, are used for various image recognition purposes. Since a large number of local descriptors are extracted from just one image, shortening the query time is crucial when handling a huge number of images. Thus, indices are indispensable in this regard for large-scale data, and as a result, various indexing methods have been proposed. In recent years, an approximate proximity search method that does not guarantee exact results has been the prevailing method used in the field because the query time rather than search accuracy is prioritized.

Hash-based and quantization-based methods are approximate searches without original objects. LSH [1], which is one of the hash-based methods, searches for proximate objects by using multiple hash functions, which compute the same

---

The original version of this chapter was revised: The presentation of Fig. 5(b) was incorrect. The erratum to this chapter is available at 10.1007/978-3-319-46759-7_26

hash value for objects that are close to each other. Datar et al. [2] applied LSH to L_p spaces so that it could be used in various applications. Spectral hashing [3] was proposed as a method that optimizes the hash function by using a statistical approach for datasets. Quantization-based methods [4,5] quantize objects and search for quantized objects. For example, the product quantization method (PQ) [5] splits object vectors into sub vectors and quantizes the sub vectors to improve the search accuracy. While recent hash-based and quantization-based methods drastically reduce memory usage, the search accuracies are significantly lower than those of proximity searches using original objects.

Proximity searches using original objects are broadly classified into tree-based and graph-based. In the tree-based method, a whole space is hierarchically and recursively divided into sub spaces. As a result, the sub spaces form a tree structure. Various kinds of methods have been proposed, including kd-tree [6], SS-tree [7], vp-tree [8], and M-tree [9]. While these methods provide exact search results, tree-based approximate search methods have also been studied. ANN [10] is a method that applies an approximate search to a kd-tree. SASH [11] is a tree that is constructed without dividing a space. FLANN [12] is an open source library for approximate proximity searches. It provides randomized kd-trees wherein multiple kd-trees are searched in parallel [12,13] and k-means trees that are constructed by hierarchical k-means partitioning [12,14].

Graph-based methods use a neighborhood graph as a search index. Arya et al. [15] proposed a method that uses randomized neighbor graphs as a search index. Sebastian et al. [16] used a k-nearest neighbor graph (KNNG) as a search index. Each node in the KNNG has directed edges to the k-nearest neighboring nodes. Although a KNNG is a simple graph, it can reduce the search cost and provides a high search accuracy. Wang et al. [17] improved the search performance by using seed nodes, which are starting nodes for exploring a graph, obtained with a tree-based index depending on the query from an object set. Hajebi et al. [18] showed that searches using KNNGs outperform LSH and kd-trees for image descriptors. Therefore, in this paper, we focused on a graph-based approximate search for image descriptors to acquire higher performance.

Let $G = G(V, E)$ be a graph, where V is a set of nodes that are objects in a d-dimensional vector space $\mathbb{R}^d$. E is the set of edges connecting the nodes. In graph-based proximity searches, each of the nodes in a graph corresponds to an object to search for. The graph that these methods use is a neighborhood graph where neighboring nodes are associated with edges. Thus, neighboring nodes around any node can be directly obtained from the edges. The following is a simple nearest neighbor search for a query object that is not a node of a graph using a neighborhood graph in a best-first manner.

An arbitrary node is selected from all of the nodes in the graph to be the target. The closest neighboring node to the query is selected from the neighboring nodes of the target. If the distance between the query and the closest neighboring node is shorter than the distance between the query and the target node, the target node is replaced by the closest node. Otherwise, the target node is the nearest node (the search result), and the search procedure is terminated.

The search performance of a KNNG improves as the number of edges for each node increases. However, the rate of improvement gradually tapers off while the edges occupy more and more memory. To avoid this problem, we propose a pruned bi-directed k-nearest neighbor graph (PBKNNG). First, it adds reversely directed edges to all of the directed edges in a KNNG. While it can improve the search performance, the additional edges tend to concentrate on some of the nodes. Such excess edges obviously reduce the search performance because the number of accesses to unnecessary nodes to search increases. Therefore, second, the long edges of each node holding excess edges are simply pruned. Third, edges that have alternative paths for exploring the graph are selectively pruned. Thus, we show that the PBKNNG outperforms not only the KNNG but also the tree- and quantization-based methods.

As the number of objects grows, the brute force construction cost of a KNNG exponentially increases because the distances between all pairs of objects in the graph need to be computed. Thus, Dong et al. [19] reduced the construction cost by constructing an approximate KNNG. Here, the ANNG [20] is not an approximate KNNG but an approximate neighborhood graph that is incrementally constructed using approximate k-nearest neighbors that are searched for by using the partially constructed ANNG. Such approximate neighborhood graphs can drastically reduce construction costs. In this paper, we also show that the search performance of a graph (PANNG) derived from an ANNG instead of a KNNG in the same way as a PBKNNG can be close to that of a PBKNNG.

The contributions of this paper are as follows.

– We propose a PBKNNG derived from a KNNG and show that it outperforms not only the KNNG but also the tree- and quantization-based methods.
– We show the effectiveness of a PANNG derived from an approximate neighborhood graph instead of a KNNG derived in the same way as a PBKNNG.

2   KNNG-Based Proximity Search

2.1   Proximity Search Algorithm

Most applications including image search and recognition require more than one object to be the result for a specific query. Therefore, we decided to focus on k-nearest neighbor (KNN) searches in this study. The search procedure with a graph-based index generally consists of two steps: obtaining seed nodes and exploring the graph with the seed nodes. Seed nodes can be obtained by random sampling [18,20], clustering [16], or finding nodes that neighbor a query by using a tree-based index [17,21]. Although the methods using a tree-based index perform the best, we used the simplest method, random sampling, in order to evaluate the graph structure without the effect of the tree-structure or clustering. As far as the second step goes, there are two methods of exploring a graph. In the first, the neighbors of the query are traced from seed objects in the best-first manner in Sect. 1, and this is done repeatedly using different seeds to improve the search accuracy [16,18]. In the second, nodes within the search
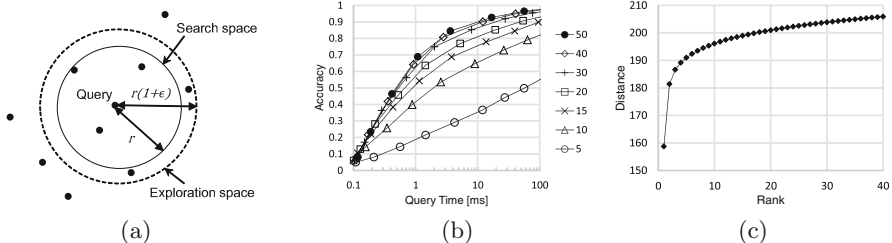
Fig. 1. (a) Relationship between the search space, exploration space, and query. (b) Search accuracy vs. query time of KNNG for different numbers of edges k for 10 million SIFT image descriptors. (c) Average distance of objects for each rank of nearest neighbors vs. rank of nearest neighbors.

space, which is narrowed down as the search progresses, are explored [17, 20]. The former method has a drawback in that the same nodes are accessed multiple times because it performs the best-first procedure repeatedly. As a result, search performance deteriorates. Therefore, we use the latter to evaluate graphs in this paper.

During KNN search, the distance of the farthest object in the search result from the query object is set as the search radius r. The actual explored space is wider than the search space defined by r. The radius of the exploration space r_e is defined as $r_e = r(1 + \epsilon)$, where ϵ expands the exploration space to improve the search accuracy. As ϵ increases, the accuracy improves; however, the search cost increases because more objects within the expanded space must be accessed. Figure 1(a) shows how the search space, exploration space, and query are related. Algorithm 1 is the pseudo code of the search. Here, KnnSearch returns a set of resultant objects R. Let q be a query object, k_s be the number of resultant objects, C be the set of already evaluated objects, $d(x, y)$ be the distance between objects x and y, and $N(G, x)$ be the set of neighboring nodes associated with the edges of node x in graph G. The function $\text{Seed}(G)$ returns seed objects sampled randomly from graph G. In a practical implementation, sets S and R are priority queues. While making set C a simple array would reduce the access cost, the initializing cost is expensive for large-scale data. For this reason, a hash set is used instead.

2.2   Problem Definition

For simplicity, we will analyze the nearest neighbor search instead of a k-nearest neighbor search. If Condition 1 is satisfied, the nearest neighbor is obtained in a best-first manner from an arbitrary node on the neighborhood graph [22].

Condition 1. $\forall a \in G, \forall q \in \mathbb{R}^d$, if $\forall b \in N(G, a), d(q, a) \leq d(q, b)$, then $\forall b \in G, d(q, a) \leq d(q, b)$.

Delaunay triangulation, which satisfies Condition 1, has absolutely fewer edges than a complete graph that also satisfies Condition 1. The number of edges,

Algorithm 1. KnnSearch

Input: G, q, k_s, ϵ
Output: R
1: $S \leftarrow \text{Seed}(G),\ r \leftarrow \infty,\ R \leftarrow \emptyset$
2: **while** $S \neq \emptyset$ **do**
3:     $s \leftarrow \underset{x \in S}{\text{argmin}}\, d(x, q),\ S \leftarrow S - \{s\}$
4:     **if** $d(s, q) > r(1 + \epsilon)$ **then**
5:       **return** R
6:     **end if**
7:     **for all** $o \in N(G, s)$ **do**
8:       **if** $o \notin C$ **then**
9:         $C \leftarrow C \cup \{o\}$
10:      **if** $d(o, q) \leq r(1 + \epsilon)$ **then**
11:        $S \leftarrow S \cup \{o\}$
12:      **end if**
13:       **if** $d(o, q) \leq r$ **then**
14:        $R \leftarrow R \cup \{o\}$
15:        **if** $|R| > k_s$ **then**
16:          $R \leftarrow R - \{\underset{x \in R}{\text{argmax}}\, d(x, q)\}$
17:        **end if**
18:        **if** $|R| = k_s$ **then**
19:          $r \leftarrow \max_{x \in R} d(x, q)$
20:        **end if**
21:       **end if**
22:      **end if**
23:     **end for**
24: **end while**
25: **return** R

however, increases drastically as the dimension of the objects increases. Therefore, a Delaunay triangulation is impractical in terms of the index size due to a huge number of the edges. As a result, most of the graph-based methods instead use a KNNG, where the number of edges can be arbitrarily specified. The search results of KNNG, however, are approximate because this graph does not satisfy Condition 1.

Figure 1(b) shows the accuracy versus query time for different numbers of edges k in a KNNG. The dataset consisted of 10 million SIFT image descriptors (128-dimensional data). The search was conducted with Algorithm 1. The curves of the figure are depicted by varying ϵ. Being closer to the top-left corner of the figure means better performance in terms of query time and accuracy. In this paper, accuracy is measured in terms of precision. In fact, precision and recall are identical in the KNN search. From Fig. 1(b), one can see that the search performance improves as the number of edges k in the KNNG increases. However, the rate of improvement gradually decreases. The memory needed for storing over 50 edges is large, whereas the improvement brought by storing so many edges is not so great.

We examined the distribution of neighboring objects around a query object. 1,000 objects were randomly selected as queries from 10 million objects, and the 40 nearest neighbors for each query object were sorted by distance. Figure 1(c) shows the average distance of the objects for each rank of the nearest neighbors. The distance of the highest ranking object that is the nearest to the query object is significantly shorter than the distances of lower ranked objects. Thus, the neighboring region around an arbitrary object is extremely sparse, while outside the neighboring region is extremely dense.

Therefore, the case in Fig. 2(a) frequently occurs in high-dimensional spaces. The figure depicts the space of distances from node o_1. The number of edges in KNNG is three. The rank of o_2 in ascending order of the distance from o_1 is much
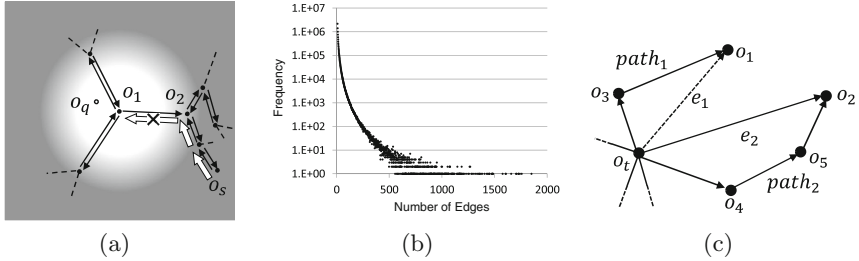
Fig. 2. (a) Relationship between nodes and edges in the case of problem conditions. (b) Frequency of nodes vs. number of edges for each node in a BKNNG. (c) Selective edge removal. The target node is o_t, which has excess edges. If $p = 3$, e_1 is removed, and e_2 is not.

higher than the rank of o_1 in ascending order of the distance from o_2. Thus, while the directed edge from o_1 to o_2 is generated, an edge from o_2 to o_1 is not generated. Therefore, during a search, when the query o_q is close to node o_1 and the seed object o_s is near object o_2, node o_1 cannot be reached through o_2 from node o_s because there is no path from o_2 to o_1. As a result, search accuracy is reduced for high-dimensional data where such conditions frequently occur. Increasing the number of edges helps to avoid such disconnections between neighboring nodes. Figure 1(b) shows that increasing the number of edges improves performance until around 30 edges, after which the improvement rate tapers off. While more edges can reduce such disconnections, more than enough edges increase the number of accessed nodes that are ineffective for searching. As a result, the query time increases.

3   Our Approach

To resolve the problem that increasing the number of edges to improve accuracy causes the query time to increase, we propose two types of graph structures: the pruned bi-directed k-nearest neighbor graph and pruned ANNG.

3.1   Pruned Bi-directed K-nearest Neighbor Graph

For a first step of our proposal, a reversely directed edge can be added for each directed edge instead of increasing the number of edges of each node. Furthermore, if a corresponding reversely directed edge already exists, it is not added. This solution can connect disconnected pairs of nodes and suppress any increase in ineffective long edges. We refer to the resultant graph as a bi-directed k-nearest neighbor graph (BKNNG). It theoretically has up to twice as many edges as a KNNG. However, since a KNNG likely has some node pairs with directed edges pointing to each other, the number of edges in a BKNNG is typically less than twice that of a KNNG. In the case of 10 million SIFT objects, the number of edges in a BKNNG generated from a KNNG wherein each node has 10 edges is

about 186 million. Therefore, the number of cases in Fig. 2(a), where one pair of nodes has one directed edge between two nodes, is about 86 million. 14 million pairs of nodes have two different directed edges between each other.

Algorithm 2. ConstructPBKNNG

Input: G, k_p, k_r, p
Output: G
1: **for all** $o \in V$ **do**
2:     **for all** $n \in N(G, o)$ **do**
3:         **if** $N(G, n) \cap \{o\} = \emptyset$ **then**
4:             $N(G, n) \leftarrow N(G, n) \cup \{o\}$
5:             **if** $|N(G, n)| > k_p$ **then**
6:                 $N(G, n) \leftarrow N(G, n) - \{\underset{x \in N(G,n)}{\arg\max}\ d(x, n)\}$
7:             **end if**
8:         **end if**
9:     **end for**
10: **end for**
11: RemoveEdgesSelectively(G, k_r, p)
12: **return** G

Algorithm 3. RemoveEdgesSelectively

Input: G, k_r, p
Output: G
1: **for all** $o \in V$ **do**
2:     **for all** $n \in N(G, o)$ **do**
3:         **if** $\mathrm{Rank}(N(G, o), n) > k_r$ **then**
4:             **if** PathExists(G, o, n, p) = $true$ **then**
5:                 $N(G, o) \leftarrow N(G, o) - n$
6:             **end if**
7:         **end if**
8:     **end for**
9: **end for**
10: **return** G

Figure 2(b) shows the frequency of nodes versus the number of edges in a BKNNG that was generated from a KNNG in which each node had 10 edges. The number of edges is widely distributed from 10 up to 1,851. The number of edges having the highest frequency is 10. The average number of edges is about 18.6. Since excess edges for some of the nodes reduce the search performance as a result of the computations for all the excess edges, the excess edges should be pruned. Too long edges of nodes holding excess edges are obviously not effective for exploring a graph because they do not connect to neighboring nodes. For a second step, to prune such edges, the edges are sorted in ascending order of length while reversely directed edges are being added. Here, let k_p be the maximum number of edges for each node after pruning. Edges whose rank is larger than k_p are forcedly removed (forced edge removal). Even though the processing cost is small enough, excess edges can be effectively reduced. Nevertheless, long and excess edges still remain. Since some of the long edges are effective for exploring the graph because they connect clusters and some are not, these edges should be selectively pruned to maintain the connections. If an edge from a source node to a destination node has an alternative path from the source node to the destination node, even if the edge is removed, the destination can be descended from the source through the path instead of the removed edge. Note that as the number of edges on the alternative path increases, the distance computation cost also increases during a search. Therefore, the shortest path should be found, and fewer edges on the path is better. For a third step, if the edges are ranked lower than k_r, where $k_r < k_p$, and have alternative paths that consist of less than p edges that should be all ranked higher than k_r for each node, they are removed (selective edge removal). Figure 2(c) shows the selective edge removal.
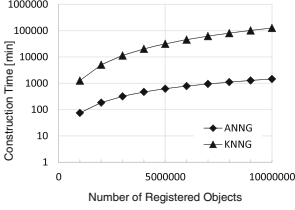
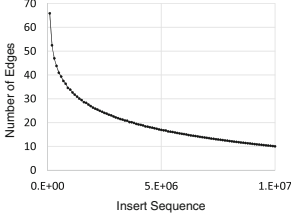**Fig. 3.** Construction time vs. number of nodes in ANNG and KNNG

Fig. 4. Average number of edges for every 100,000 consecutive nodes of ANNG

Algorithm 4. ConstructPANNG

Input: $O, k_c, k_p, k_r, \epsilon_c, p$
Output: G
1: **for all** $o \in O$ **do**
2:     $N(G, o) \leftarrow \text{KnnSearch}(G, o, k_c, \epsilon_c)$
3:     **for all** $n \in N(G, o)$ **do**
4:         $N(G, n) \leftarrow N(G, n) \cup \{o\}$
5:         **if** $|N(G, n)| > k_p$ **then**
6:             $N(G, n) \leftarrow N(G, n) - \{\underset{x \in N(G,n)}{\text{argmax }} d(x, n)\}$
7:         **end if**
8:     **end for**
9: **end for**
10: RemoveEdgesSelectively(G, k_r, p)
11: **return** G

$Path_1$ is the shortest alternative path of the edge e_1. $Path_2$ is the shortest alternative path of e_2. If $p = 3$, then e_1 is removed because the number of edges on $path_1$ is two. However e_2 is not removed because that on $path_2$ is three. Although finding the shortest path is time consuming, the limitation p of the number of edges on the alternative paths contributes to reducing the processing time to find the shortest alternative path. We refer to the resultant graph as a pruned bi-directed k-nearest neighbor graph (PBKNNG). Algorithm 2 shows the pseudo code for constructing a PBKNNG. Here, a KNNG is the input graph $G = G(V, E)$. Algorithm 2 calls Algorithm 3, which is the selective edge removal. Rank$(N(G, o), n)$ returns the rank of a node n by the distance (edge length) to the neighboring nodes $N(G, o)$ of a node o. PathExists(G, o, n, p) exhaustively explores the graph G from o within p edges and returns whether the shortest alternative path from o to n exists.

3.2   Pruned ANNG

While the KNNG is extremely expensive to construct, an approximate neighborhood graph is much less costly. An ANNG [20], which is one of the approximate neighborhood graphs, has high search performance. To create an ANNG incrementally, approximate k-nearest neighbor objects for edges are searched for by using the partially created ANNG. Figure 3 shows construction times for KNNG and ANNG. KNNG construction times for more than two million objects were estimated from the construction time for one million objects. The figure shows that an ANNG has significantly lower construction times compared with a KNNG. However, the initially inserted nodes of the ANNG tend to have a huge

number of edges compared with the subsequently inserted nodes. Figure 4 shows
the average number of edges for every 100,000 nodes along the insertion sequence
of 10 million SIFT image descriptors, wherein 10 nearest neighbors are added as
edges for each node during insertion. The number of edges for the first sequence
exceeds 60. The excess edges are pruned in the same way as in PBKNNG, and
we refer to the resultant graph as a pruned ANNG (PANNG). Algorithm 4 is
the pseudo code for creating a PANNG. Let O be the set of inserted objects and
k_c be the initial number of edges for an inserted node. ϵ_c is for the expansion
factor of the explored space of the KNN search. KnnSearch(G, o, k_c, ϵ_c) is a KNN
search function that returns the k_c nearest neighbors to the query object o. In
this study, Algorithm 1 is also used as the KnnSearch in Algorithm 4.

4   Experimental Results

The experiments used 128-dimensional SIFT image descriptors [23] and 960-
dimensional GIST image descriptors [24]. SIFT is a local descriptor, and GIST
is a global descriptor. The descriptors were extracted from about 1 million images
downloaded from Flickr[1]. The SIFT descriptors were extracted from the image
set by using OpenCV[2]. Since just one GIST descriptor is extracted from an
image by using Lear's GIST C implementation[3], the GIST descriptors were
extracted from 4 by 4 block images into which each image in the image set
was divided in order to extract 10 million descriptors. Duplicates were removed
from the descriptors. A 10-million-object dataset and 500-object query set were
randomly selected from each of the descriptors. A Euclidean distance function
was used. Each SIFT element was stored in memory as a 1-byte integer, and
each GIST element was stored as a 4-byte floating point number. The resulting
size of the KNN search was 20 nearest neighbors. We conducted the experiments
on an Intel Xeon E5-2630L (2.0 GHz and 64 GB of memory). Although the CPU
had six cores, the experimental software was not processed in parallel.

Parameter Determination: First, we evaluated the search performance to
determine the number of seed nodes. The search performance was assessed in
terms of the query time and the search accuracy while varying the number of the
seed nodes from 1 to 100 using the SIFT dataset. The results indicated that the
query times for all seed node numbers were almost the same when the accuracy
was over 0.5. The query time for 10 seed nodes was slightly shorter when the
accuracy was less than 0.5. Thus, 10 seed nodes were used in the experiment.

Figure 5(a) plots the search performance of a BKNNG using the SIFT dataset
with a varying number of edges k_o, which represents not the actual number of
edges but the number of edges for the original KNNG from which the BKNNG is
derived. It can be seen that excess edges tended to reduce performance, and the
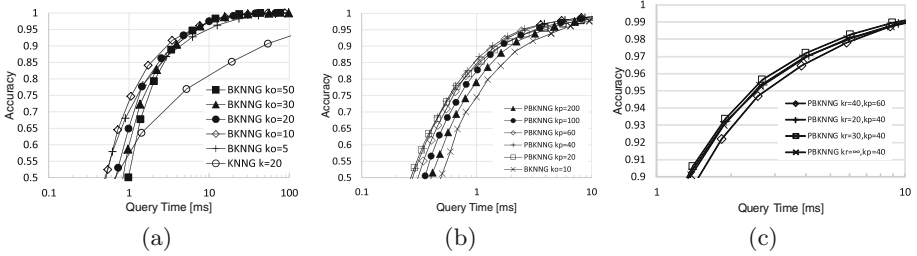plot for $k_o = 10$ indicates that 10 edges gave the best performance almost overall.

Fig. 5. Accuracy vs. query time for SIFT dataset. (a) BKNNG for different numbers of edges k_o and BKNNG with $k_o = 10$. (b) PBKNNGs with $k_r = \infty$ derived from BKNNG with $k_o = 10$ for different values of k_p. (c) PBKNNGs derived from BKNNG with $k_o = 10$ for different values of k_p and k_r.

Since search performance largely depends on the number of edges, to equitably compare the different graphs, the total numbers of edges in the graphs should be as close to equal as possible. The total number of edges in the BKNNG is up to twice that of the original KNNG. Therefore, to compare them, Fig. 5(a) also shows the performance for a KNNG with $k = 20$. The KNNG and the BKNNG with $k_o = 10$ had almost identical numbers of edges. The actual average number of edges in the BKNNG was about 18.6 because edges were not added to node pairs that already had two different directed edges between them. In spite of it having fewer edges than the KNNG, the BKNNG performed considerably better than the KNNG in a higher accuracy range. For example, the query time of the BKNNG was more than 10 times shorter than that of the KNNG at an accuracy of 0.9. These results indicate that adding bi-directed edges to the KNNG significantly improved performance in this range of accuracy. Figure 5(b) shows the performance of PBKNNGs with $k_r = \infty$ derived from a KNNG with $k_o = 10$ for different values of k_p. The parameter $k_r = \infty$ disables the selective edge removal. The BKNNG has many edges, and these edges increase the query time. Therefore, while pruning edges improves performance, pruning too many edges reduces it. From Fig. 5(b), it can be seen that the PBKNNGs where k_p is 20 and 40 show almost identical performance and are better than the others. Figure 5(c) shows the performance of PBKNNGs derived from KNNG with $k_o = 10$ and $p = 3$ for different values of k_r and k_p in a higher accuracy range. Since the performance obviously decreased where $p > 3$, we adopted $p = 3$ in all of the experiments. The PBKNNG with $k_p = 40$ and $k_r = 30$ was slightly better than the others. This shows that selective edge removal is more effective at improving performance.

Figure 6(a) shows the performance of PANNG[4] for different values of k_p and k_r. The PANNG was constructed by using Algorithm 4 with $k_c = 10$ and $\epsilon_c = 0.1$. The curves in the figure show a similar tendency to those for PBKNNG in Fig. 5(b) and (c). The nodes inserted in the initial stage tend to have a huge number of edges, as Fig. 2(b) shows. Therefore, pruning contributes to improving search performance. However, pruning too many edges reduces performance in
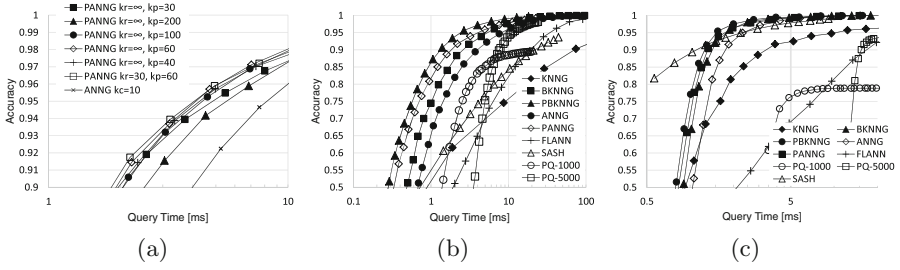
---

[4] http://research-lab.yahoo.co.jp/software/ngt/.

(a)   (b)   (c)

Fig. 6. Accuracy vs. query time. (a) ANNG with $k_c = 10$ and PANNGs derived from the ANNG for different k_p and k_r for SIFT. (b) Comparison of KNNG ($k = 20$), BKNNG ($k_o = 10$), PBKNNG ($k_r = 30$, $k_p = 40$), ANNG ($k_c = 10$), PANNG ($k_r = 3$, $k_p = 60$), FLANN, SASH, PQ ($R = 1000$), and PQ ($R = 5000$) for SIFT. (c) Comparison of KNNG ($k = 20$), BKNNG ($k_o = 10$), PBKNNG ($k_r = 30$, $k_p = 40$), ANNG ($k_c = 10$), PANNG ($k_r = 30$, $k_p = 60$), FLANN, SASH, PQ ($R = 1000$), and PQ ($R = 5000$) for GIST.
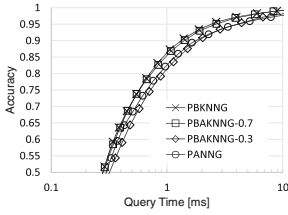

Fig. 7. PBKNNG, PBAKNNGs, and PANNG with various edge precisions indicated in Table 1 for SIFT

Table 1. Average edge precision and average rank of top 10 shortest edges for PBKNNG, PBAKNNGs, and PANNG

Graph	Average precision	Average rank
PBKNNG	1.00	5.50
PBAKNNG-0.7	0.706	7.79
PBAKNNG-0.3	0.303	17.9
PANNG	0.567	29.4

the same way as with PBKNNG. From the figure, the PANNG with $k_p = 60$ and $k_r = 30$ performed the best overall in a higher accuracy range.

Comparison of Graph-Based Indexes, FLANN, SASH, and PQ: Figures 6(b) and (c) compare the performances of KNNG, BKNNG, PBKNNG, ANNG, PANNG, FLANN, SASH[5], and PQ for SIFT and GIST using the determined parameters. KNNG performed the worst, and PBKNNG performed the best among the graph-based methods. Although PANNG was slightly worse than PBKNNG, it is practically advantageous because the construction cost of an ANNG is considerably lower than that of an exact KNNG.

FLANN automatically selects the best algorithm for the dataset. For the SIFT and GIST dataset, it selected hierarchical k-means partitioning. For constructions of SASH, we used the number of parent nodes $p = 4$. Even though approximate searches without original objects are not our target, we compared it with PQ. While PQ does not require the objects in the memory, the search accuracy is significantly lower. To compare fairly, we added a verification step

---

[5] http://research.nii.ac.jp/~meh/sash/.

after the PQ search, which computes distances for the results of the PQ using the objects in the memory and returns the k nearest neighbors. According to the experiment of PQ [5], the best parameters were explored and determined. We used the number of codewords for the product quantization $k^* = 256$, the number of subvectors $m = 8$, and the number of codewords $k' = 1024$. The curves of PQ were plotted by varying the number of the nearest neighbors of the coarse quantizer w for the number of the nearest neighbors of PQ $R = 1000$ and 5000. It can be seen that PBKNNG and PANNG outperformed FLANN and PQs overall and outperformed SASH excluding at lower accuracy for GIST in Fig. 6(b) and (c).

Edge Precision Effect Analysis: In spite that a KNNG does not satisfy Condition 1, the PBKNNG derived from the KNNG works well. This suggests that it might be unnecessary to use an exact KNNG to generate a PBKNNG. To clarify this, just as we derived the PBKNNG from the KNNG, we derived a pruned bi-directed approximate k-nearest neighbor graph (PBAKNNG) from an approximate KNNG, which is intentionally generated by pruning the edges of the KNNG according to a specific probability, called "edge precision." Fig. 7 compares the performances of PBKNNG, PBAKNNGs, and PANNG to clarify the effect of varying the edge precision. The PBKNNG was constructed with $k_o = 10$. The PBAKNNGs were derived from AKNNG with $k_o = 10$ for the edge precisions 0.7 and 0.3. The PANNG was constructed with $k_c = 10$ for $\epsilon_c = 0.1$. All of them were constructed with $k_p = 40$ and $k_r = 30$. Table 1 shows the average edge precision and the average rank of the top 10 shortest edges for each of 1,000 randomly sampled nodes of the indexes in Fig. 7. From the order of the average precisions in Table 1, the search performances of PANNG should be between PBAKNNG-0.7 and PBAKNNG-0.3. It is, however, almost the same as PBAKNNG-0.3 at higher accuracy. We suppose that performance is affected by both the precision and the average rank of edges. Since the performance decreases for low edge precision are all rather small, these results show that an exact KNNG is dispensable in order to make an approximate search.

Memory Usage: Since our search algorithm needs a large number of distance computations, all of the objects should be placed in memory to reduce the search cost. Here, we will discuss the memory usage of a logical index structure instead of an actual structure since our actual implementation uses a standard template library (STL) including a non-negligible amount of memory overhead. The logical index structure has an array of nodes consisting of objects, a pointer to the edge array for each node, and the size of the edge array. Its memory usage is as follows.

$$
\begin{aligned}
\text{memory usage} &= \text{node array usage} + \text{edge array usage} \\
\text{node array usage} &= (\text{object dimensionality} \cdot \text{size of object element variable} \\
&\quad + \text{size of pointer to edge array} \\
&\quad + \text{size of edge array size variable}) \cdot \text{total number of objects} \\
\text{edge array usage} &= \text{size of node ID variable} \cdot \text{total number of edges}
\end{aligned}
\tag{1}
$$

The length of each edge is used to prune excess edges, so the memory usage for the edge array for index construction is as follows.

$$\text{edge array usage} = (\text{size of node ID variable} + \text{size of distance variable}) \cdot \\ \text{total number of edges} \tag{2}$$

The total numbers of edges for the PBKNNG for $k_o = 10$, $k_p = 40$, and $k_r = 30$ are 165,529,883 for the SIFT dataset and 163,959,473 for the GIST dataset. The logical memory usage derived from Formula 1 amounts to 1.90 GB for SIFT and 36.5 GB for GIST. Since the experimental implementation included additional information for the evaluations and memory overhead of the STL, the actual amount of the memory was not measured.

5   Conclusion

We derived a PBKNNG from a KNNG as an index of high-dimensional data such as image descriptors. The experiment showed that the PBKNNG outperforms not only the KNNG but also the FLANN, SASH, and PQ in most cases on SIFT and GIST datasets. The drawback of the KNNG is its high construction cost, and an approximate neighborhood graph is much less costly. The experiment also showed a PANNG derived from the approximate neighborhood graph instead of a KNNG in the same way as the PBKNNG outperforms the KNNG, FLANN, SASH, and PQ in most cases and performs only a little worse than the PBKNNG.

References

1. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Proceedings of 25th International Conference on Very Large Data Bases, pp. 518–528 (1999)
2. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the 20th Annual Symposium on Computational Geometry, pp. 253–262. ACM (2004)
3. Weiss, Y., Torralba, A., Fergus, R.: Spectral hashing. In: Advances in Neural Information Processing Systems, pp. 1753–1760 (2009)
4. Gong, Y., Lazebnik, S.: Iterative quantization: a procrustean approach to learning binary codes. In: 2011 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 817–824. IEEE (2011)
5. Jegou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. IEEE Trans. Pattern Anal. Mach. Intell. **33**(1), 117–128 (2011)
6. Bentley, J.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**, 509–517 (1975)
7. White, D., Jain, R.: Similarity indexing with the SS-tree. In: Proceedings of 12th International Conference on Data Engineering, pp. 516–523 (1996)
8. Yianilos, P.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 311–321 (1993)

9. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: Proceedings of International Conference on Very Large Data Bases, pp. 426–435 (1997)
10. Arya, S., Mount, D., Netanyahu, N., Silverman, R., Wu, A.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. J. ACM **45**(6), 891–923 (1998)
11. Houle, M.E., Sakuma, J.: Fast approximate similarity search in extremely high-dimensional data sets. In: 21st International Conference on Data Engineering (ICDE 2005), pp. 619–630. IEEE (2005)
12. Muja, M., Lowe, D.G.: Scalable nearest neighbor algorithms for high dimensional data. IEEE Trans. Pattern Anal. Mach. Intell. **36**(11), 2227–2240 (2014)
13. Silpa-Anan, C., Hartley, R.: Optimised KD-trees for fast image descriptor matching. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2008, pp. 1–8. IEEE (2008)
14. Nister, D., Stewenius, H.: Scalable recognition with a vocabulary tree. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, vol. 2, pp. 2161–2168. IEEE (2006)
15. Arya, S., Mount, D.M.: Approximate nearest neighbor queries in fixed dimensions. In: Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA 1993, Philadelphia, PA, USA, pp. 271–280. Society for Industrial and Applied Mathematics (1993)
16. Sebastian, T., Kimia, B.: Metric-based shape retrieval in large databases. In: Proceedings of 16th International Conference on Pattern Recognition, vol. 3. 291–296 (2002)
17. Wang, J., Li, S.: Query-driven iterated neighborhood graph search for large scale indexing. In: Proceedings of the 20th ACM International Conference on Multimedia, MM 2012, pp. 179–188. ACM, New York (2012)
18. Hajebi, K., Abbasi-Yadkori, Y., Shahbazi, H., Zhang, H.: Fast approximate nearest-neighbor search with k-nearest neighbor graph. In: Proceedings of the 22nd International Joint Conference on Artificial Intelligence, pp. 1312–1317 (2011)
19. Dong, W., Moses, C., Li, K.: Efficient k-nearest neighbor graph construction for generic similarity measures. In: Proceedings of the 20th International Conference on World Wide Web, WWW 2011, pp. 577–586. ACM, New York (2011)
20. Iwasaki, M.: Proximity search in metric spaces using approximate k nearest neighbor graph (in Japanese). IPSJ Trans. Database **3**(1), 18–28 (2010)
21. Iwasaki, M.: Proximity search using approximate k nearest neighbor graph with a tree structured index (in Japanese). IPSJ J. **52**(2), 817–828 (2011)
22. Navarro, G.: Searching in metric spaces by spatial approximation. VLDB J. **11**(1), 28–46 (2002)
23. Lowe, D.G.: Object recognition from local scale-invariant features. In: The Proceedings of the Seventh IEEE International Conference on Computer Vision, vol. 2, pp. 1150–1157. IEEE (1999)
24. Oliva, A., Torralba, A.: Modeling the shape of the scene: a holistic representation of the spatial envelope. Int. J. Comput. Vis. **42**(3), 145–175 (2001)