Stratified Graph Indexing for Efficient Search in Deep Descriptor Databases

M M Mahabubur Rahman[1] and Jelena Tešić[1*]

[1]Computer Science, Texas State University, 601 University Dr, San Marcos, 78666, Texas, USA.

*Corresponding author(s). E-mail(s): jtesic@txstate.edu;
Contributing authors: toufik@txstate.edu;

Abstract

Searching for unseen objects in extensive visual archives is challenging, demanding efficient indexing methods that can support meaningful similarity retrievals. This research paper presents the Stratified Graph (SG) approach for indexing similar deep descriptors by sorting them into distance-sensitive layers. The indexing algorithm incrementally constructs a bi-directional $\boldsymbol{m}$-nearest neighbor graph within each layer, with additional $\mathbf{1}$-nearest neighbor links from outer layers, providing a distant scaling property in the graph structure. The search process starts from the innermost layer, and the same layer neighbors enhance Average Recall (AR), while the distant scaling property enhances search speed, maintaining logarithmic complexity scaling. We compare and contrast SG with six state-of-the-art retrieval methods in four deep-descriptor and two classical-descriptor databases, and we show that the SG indexing and search has smaller memory usage (up to four times) and the Mean Average Precision and AR improve up to 8% over state-of-art for all six datasets at five retrieval depths.

Keywords: Similarity search, High-dimensional indexing, Deep descriptors search, Information retrieval, Vector search, Graph-based index

1 Introduction

Video archives house immense data and analytics that cannot be processed manually. The task of identifying similar objects in exabytes of video archive data in a scalable and effective way can help resolve many video analytics tasks, for example, the surveillance task: *When was this object spotted before?*; the crowd sensing task *Was this object in the archive footage?*, and the reconnaissance task: *Find me similar objects in archived footage.*

Classifiers sparsely help as object labeling is sparse, and labels are unavailable for every target of interest. Deep descriptors capture objectness characteristics well for the long tail of tasks, regardless of the size or density of the objects, as shown in [1, 2] for overhead imagery. Recent object detectors built on CenterNet2[3], YOLOv4 [4], SOD [5], and TPH-YOLOv5 [6] capture the objectness in a feature vector well. Now, identifying similar objects in exabytes of video archives reduces to designing an efficient indexing and search system in the deep descriptor databases [7]. This research proposes a novel indexing and search approach that supports efficient and effective *Approximate Nearest Neighbor* search in high-dimensional deep descriptor space.

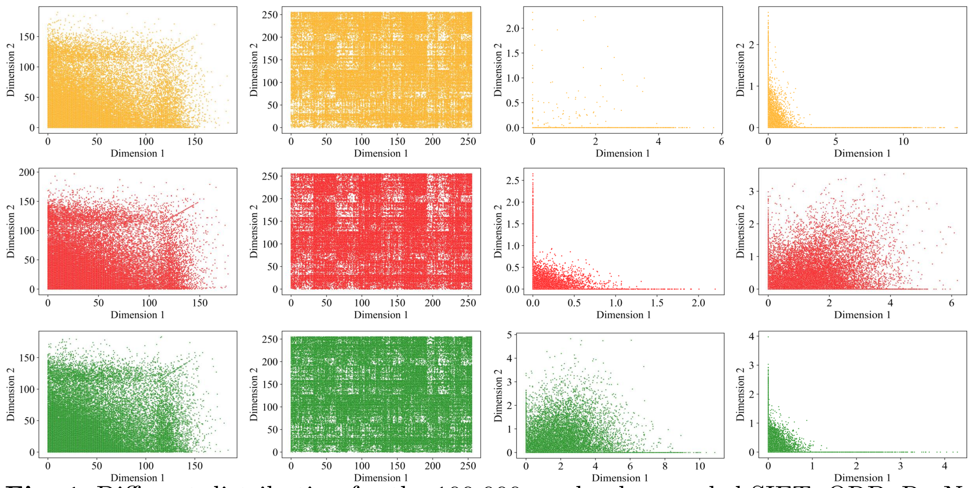

Fig. 1: Different distribution for the 100,000 randomly sampled SIFT, ORB, ResNet, and Mobile V3 descriptor values (left to right) for the first two dimensions of the VisDrone (top), DOTA (middle), and DIOR (bottom) dataset.

1.1 Motivation

The dissimilarity between deep descriptor databases and conventional descriptors necessitates particular consideration within the context of Approximate Nearest Neighbor (ANN) search applications. To elucidate the disparities between deep descriptors and their traditional counterparts, this research undertakes the extraction of features from three experimental datasets. The conventional descriptors, namely Scale-Invariant Feature Transform (SIFT) [8] and Oriented FAST and Rotated BRIEF (ORB)[9], are employed as the feature extraction pipeline for the traditional descriptors. In contrast, deep descriptors are generated using the ResNet-50 [10] and MobileNetV3[11] pipelines. Table 1 shows the characteristics of the feature databases for different feature extractors. Deep descriptor databases are high dimensional (1024) and sparse: for example, on average *65.86%* zeros per feature vector in the deep descriptor datasets compared to the traditional feature descriptors which have only *6.69%* zeros. Thus, the sparsity and high dimensionality curse will degrade the similarity search performance compared to conventional descriptor databases with a low percentage of zero entries and smaller dimensions. Figure 1 illustrates the value distribution of feature vectors and how different traditional features are from the deep features.

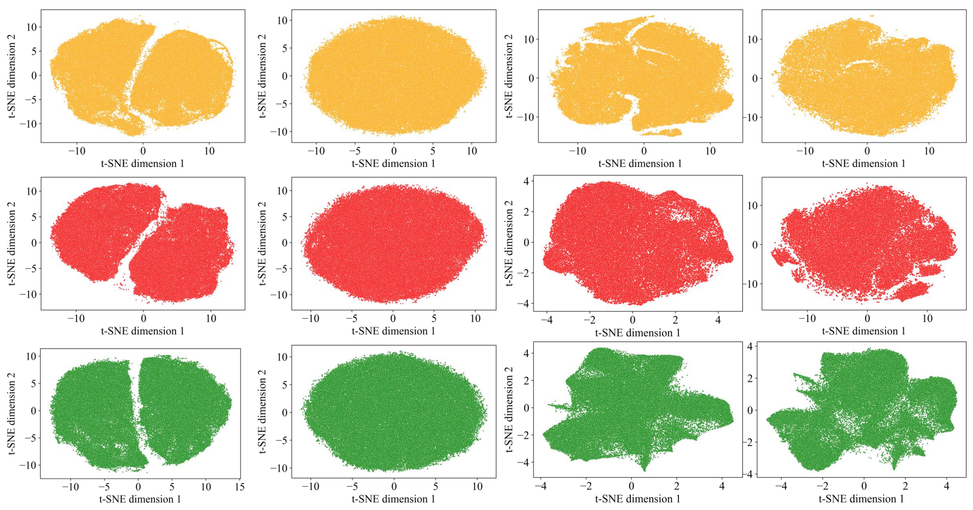

Fig. 2: t-SNE distribution for 100,000 VisDrone (top), DOTA (middle), and DIOR (bottom) SIFT, ORB, ResNet, and Mobile V3 descriptor values (left to right)

Deep descriptors exhibit a concentration of feature vector values near the origin. This concentration phenomenon implies that the dataset shows high sparsity within the feature space, whereby a substantial portion of the feature vectors appears confined to a specific region along the primary axis. Understanding the sparsity patterns in the feature space can provide valuable insights for optimizing ANN search processes on deep descriptors. In Figure 2, we present t-SNE distributions of random 100,000 vectors from deep descriptor databases, while Figure 2 displays the t-SNE distribution of traditional descriptors. The conventional SIFT and ORB descriptors show a similar distribution for all three datasets in Figure 2 where the features are clustered in two dense clusters for SIFT. For ORB, the features are clustered into a single dense cluster. The deep descriptors show varying distributions for different datasets in Figure 2.

Table 1: SIFT and ORB, Resnet50 and MobilenetV3 descriptor datasets.

Descriptor	SIFT		ORB		ResNet50		MobileNetV3	
Dimension [Source]	128 [8]		32 [9]		1024 [10]		1024 [11]	
Dataset	# in Ms	% of 0s	# in M	% of 0s	# in M	% of 0s	# M	% of 0s
VisDrone (Videos) [12]	5.28	15.13	1.29	0.54	0.12	63.37	0.13	80.97
DOTA2.0 (Images) [13]	3.40	12.05	0.92	0.73	0.16	56.28	0.24	79.54
DIOR (Images) [14]	2.89	11.21	5.81	0.53	0.11	46.91	0.17	68.11

The deep descriptors tend to cluster into different regions and exhibit more discrimination for similarity search than traditional descriptors. Given our objective of effectively and efficiently finding similar objects represented by high-dimensional sparse deep descriptors, our focus is on proposing an indexing and search structure for deep descriptor databases. Similarity search looks for object representations in a database that are similar or close to a query based on a specific similarity measure. That measure is usually a distance function. Let's define X as a metric space with associated distance function $d(p, q)$, and P as a set of points in that metric space $p, q \in P$. The *nearest neighbor* of a query point q is p if $d(p, q) \leq d(q, p'), \forall q, p, p' \in P, p \neq p'$. The k-nearest neighbors (k-NN) search identifies the top k nearest neighbors to query

q and has complexity $O(|P| \times d)$, where $|P|$ is a number of points in P and d is the dimension of points in P. Thus, k-NN does not scale for $|P|$ in millions or billions and d in hundreds and thousands, and the search task becomes indexing and search task for large high-dimensional descriptor databases.

2 State Of The Art

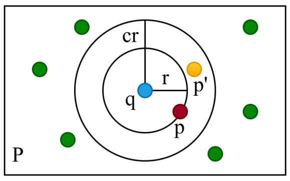

Fig. 3: Approximate Nearest Neighbor search: p(red dot) is the nearest neighbor, and p'(orange dot) is the approximate nearest neighbor to query q (blue dot).

Approximate Nearest Neighbor Indexing and Search: The index is the data structure that summarizes point dataset P and allows for the efficient approximate nearest neighbor retrieval that closely matches brute force nearest neighbor retrieval set without the need to compute all distances from query vector q to all the points in P. The approximate nearest-neighbor (ANN) methods are optimized to balance efficiency and effectiveness and offer speed up at the account of the accuracy [15]. ANN methods are optimized to return *any* point $p' \in P$ such that the distance from q to p' is at most $c \cdot min_{p \in P} D(q, p)$, for some $c \geq 1$. In Figure 3, the distance to the nearest neighbor $min_{p \in P} D(q, p)$ from query q is shown as r. Therefore, the approximate nearest neighbor p' from query q is within distance cr.

Related Work: State of the art can be roughly grouped as graph-based [16–19], hashing-based [20–22], and partition-based [23, 24] methods. The *Faiss* library[8] enables efficient partitioning of data in Voronoi cells [8], where the index of each cell is a centroid of that cell and product quantization is used [25] to compress data. Annoy[23] generates several hierarchical 2-means trees by recursive partitioning. Each iteration forms two centers by conducting a basic clustering algorithm on a subset of samples from the input data points. The two centers define a partition hyperplane that is equidistant from each other. The hyperplane then partitions the data points into two sub-trees, and the algorithm iteratively generates the index on each sub-tree. This approach did not scale to larger high dimensional datasets in terms of the speed of recovery and accuracy of the retrieved results [26]. The research paper [27] introduces a novel approach to graph partitioning through supervised learning. The core concept involves creating a k-NN graph and then dividing it into predefined segments of roughly equivalent size while minimizing inter-segment edges. Next, the optimal partition is achieved by training a classifier on data points, with their labels denoting the segments determined in the earlier step. This method proves to be particularly effective for managing index storage within a distributed system.*Relative-Nearest Neighbor Descent* (RNN-Descent) [19] merges NN-Descent, an algorithm for constructing approximate k-NN graphs, with RNG-Strategy, an edge selection algorithm. However, RNN-Descent focuses on the speed of index creation and loses some accuracy in the process. *Hierarchical Navigable Small World* (HNSW) [28] arranges the graph into a hierarchy of proximity graph layers with a lower layer containing all the feature vectors and higher layers containing a subset of previous layers in the hierarchy. However, this architecture results in a larger index size shown in Section 5.

Navigating Spread-out Graph (NSG) [29] favors the "Navigating Node" to make the search efficient. Still, due to high indexing complexity, NSG does not scale well when the dimension of the feature vector grows [17]. *Navigating the satellite system graph* (NSSG) improves over SSG as it introduces the satellite system graph (SSG) and a more efficient pruning technique during index building to address the high-dimensional curse. A parameter can control the sparsity of NSSG, but the chances that the monotonic search stage fails are more significant as the size and dimension of the database increases [17]. The *Tree-Based Search Graph* (TBSG) proposes to handle this problem with the probability of monotonic search success by combining the Cover Tree [30] and BKNNG (Bi-directed K-Nearest Neighbor Graph) [31] algorithms [32]. In the *Hierarchical Satellite System Graph* (HSSG), the nodes in the dataset are separated into layers, and NSGs are created on each layer separately. When searching in the high layer, the search process can skip a long distance, reducing the number of steps in extensive data [18]. The index processes in separate layers are independent once the nodes are picked, and the index algorithm can be run distributively, decreasing the index algorithm's time consumption. HSSG performs a faster coarse search on the upper layer with fewer nodes during the search. Following the coarse finds, HSSG conducts a more precise recursive OK search in the bottom layer at the cost of the high indexing and memory overhead compared to SSG [18].

Neighborhood Graph Tree (NGT) [33] uses a range search during the graph construction mechanism, and, to avoid a high degree of neighboring nodes and reduce memory overhead, applies a three-degree adjustment by connecting each feature vector to its three nearest neighbors throughout the graph. During the query process, NGT generates a seed using the VP tree [34] and performs a range search to obtain the nearest neighbors. A significant drawback of NGT is that if the query and seed are far from each other in the search space, it takes many hops between to reach the query from the root, and thus increases the retrieval time. One way to address this problem is to transform the k nearest neighbor graph into an undirected one, and the other is to construct an undirected graph by continuously inserting elements [35].

***L**earned **I**ndex for large-scale **DE**nse passage **R**etrieval* (LIDER) 's [36] hierarchical architecture is based on clustering and consists of two levels of core models. A core model is the basic unit of LIDER for indexing and searching data. It consists of an adapted Recursive Model Index (RMI) and a dimension reduction component that contains an expanded SortingKeys-LSH (SK-LSH) and a critical re-scaling module. High-dimensional dense embeddings are converted into one-dimensional keys and sorted to make quick predictions by the RMI. However, for a small number of clusters, each cluster yields many feature vectors, making it more difficult for RMI to learn the distribution effectively. As a result, the quality of in-cluster retrieval degrades, and for many clusters, the Recall suffers. The paper [37] capitalizes on contemporary hardware architectures to enhance the efficiency of approximate nearest neighbor (ANN) search, and it shows that leveraging the collective computational power of multiple cores and a heterogeneous memory system can improve performance in large-scale vector search.

Stratified Graph Approach: We propose the *Stratified Graph (SG)* indexing (SGI) and retrieval (SGR) algorithms to address the challenges of searching through

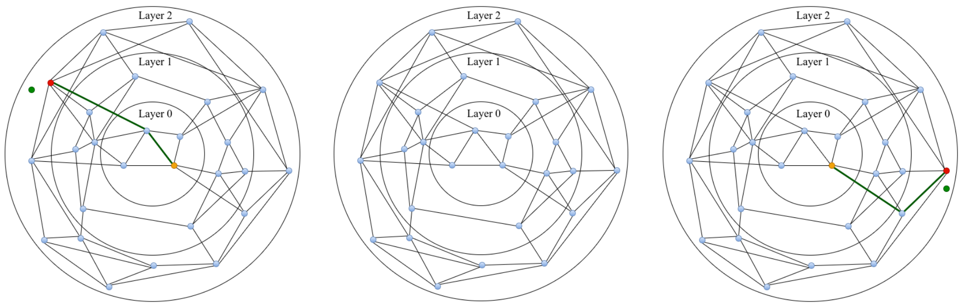

Fig. 4: Illustration of SG index (left) and two different scenarios (center and right) in a Stratified Graph (SG) search. Orange denotes the starting point of a search, red denotes the nearest neighbor, and green edges show the path of the greedy algorithm to the query(shown in green).

high-dimensional and sparse datasets of deep descriptors. It has been shown that state-of-the-art neighboring ANN methods suffer from a long index-building time and low Recall for large deep-descriptor databases [26]. This paper compares and contrasts the new stratified graph approach with state-of-the-art for large sparse descriptor retrieval.

The Stratified Graph Indexing (SGI) addresses the dataset's sparsity by centering layers close to the dataset's center of mass. The index is constructed in each layer as a bi-directional graph and designed to achieve skip list properties, as presented in detail in Section 3. Section 4 offers the novel search and retrieval approach. The advantage of the Stratified Graph is that it does not have a hierarchical structure and that SG connectivity at the same layer enhances the Recall or the ability to retrieve relevant results. Note that the SG connections between layers help maintain logarithmic complexity scaling for faster search speed. We present an in-depth comparison of multiple state-of-the-art methods over six data collections and compare the performance of the indexing and search methods in the word embedding, visual descriptor, and three deep descriptor databases in terms of high average Recall and mean average Precision at any depth of retrieval and fast retrieval times, and index size in Section 5. The research paper is concluded with a summary and next steps in Section 9.

3 Stratified Graph Indexing (SGI)

The Stratified Graph Indexing (SGI) arranges the feature vectors into layers based on their distances from the centroid. The center of the target is computed as the average of the sample or the entire dataset P. The feature vectors closer to the centroid are stratified in closer layers, while the feature vectors farther away are stratified in farther layers. Figure 4(left) illustrates the Stratified Graph Indexing (SGI) for the Euclidean distance, and the layers are stacked into the target shape in 2D. Note that the layers can be illustrated as a set of rotated squares in 2D illustration for the Manhattan distance. The bidirectional graph is constructed by connecting each feature vector to its m-Nearest Neighbors. We specify the maximum number of layers l_{max} as $log_2 m + 1$. The layer numbers determine the layer width by dividing the distance to the farthest point from the center. Therefore, all the layers have the same width. The index building starts from the outermost layer where all the m neighboring connections for each feature are within the same layer l_{max}. In layer $l_{max} - 1$, each feature vector is connected to $m - 1$ neighbors within layer $l_{max} - 1$ and 1 neighbor with layer l_{max} totaling m neighbors for each feature vectors. Therefore, at any layer

l, the number of in-layer connections is determined by $m - (l_{max} - l)$, and next-layer connections are chosen by $(l_{max} - l)$. The subsequent layer connections allow the SGI algorithm to achieve skip list properties, which helps the Stratified Graph Retrieval (SGR) algorithm a faster search in the graph, which we discuss later in this section. Higher vertex degree m leads to higher index size, slower retrieval, but higher Recall as each point is connected to its actual m nearest neighbors.

Algorithm 1: LAYERING $\big(X, M, f\big)$

```
Input: data vector $P$, number of established connections $m$, outlier filtering factor $f$
Output: element list with assigned layer
1  $numLayer \leftarrow \lfloor \log_2 m \rfloor$ ;
2  $cen \leftarrow$ mean of $X$ ;
3  $dist \leftarrow$ distances from the centroid to all data vectors ;
4  $avg \leftarrow$ mean of all distances ;
5  $\sigma \leftarrow$ standard deviation of all distances ;
6  $u_b \leftarrow avg + f \times \sigma$ ;
7  $l_b \leftarrow$ smallest of $dist$ ;
8  $r \leftarrow \frac{u_b - l_b}{numLayer}$ ;
9  $layeredElem \leftarrow \phi$ ;
10 foreach $\big(d, p\big)$ of $\big(dist, P\big)$ do
11     $l \leftarrow \lfloor \frac{d}{r} \rfloor$ ;
12     add element $p$ to layer $l$ in $layeredElem$ ;
13 end
14 return $layeredElem$ ;
```

Algorithm 2: SGI $\big(SGI, P, m, f, cand\big)$

```
Input: stratified graph index $SGI$, dataset $P$, graph degree $m$, outlier filtering factor $f$, size of dynamic candidate list $cand$
Output: Updated $SGI$ inserting all the elements in $P$
1  $graph \leftarrow \phi$ ;
2  $layerGraphList \leftarrow \phi$ ;
3  $layeredElem \leftarrow$ LAYERING$(P, m, f)$ ;
4  foreach $layer$ of $layeredElem$ do
5      $clg \leftarrow \phi$ ;
6      foreach $elem$ of $layer$ do
7          $clg \leftarrow$ ADD$(clg, elem, m, cand)$ ;
8          if $layerGraphList$ not empty then
9              foreach $g$ in $layerGraphList$ do
10                 $n \leftarrow$ SGR$(g, elem, k = 1, cand)$ ;
11                 $clg \leftarrow$ update $clg$ inserting $n$ to the neighbor list of $elem$ ;
12             end
13         end
14         add $clg$ to $layerGraphList$ ;
15     end
16     $m \leftarrow m - 1$ ;
17 end
18 $graph \leftarrow \cup layerGraphList$ ;
```

The index-building phase involves determining the layers of each element based on their distances from the centroid and then constructing a kNN graph within each layer, from the outermost layer to the innermost layer. The outlier filtering factor f filters out elements too far from the mean distance and ensures that outliers do not affect the layer boundary. Algorithm 1 describes the process of determining the layers for each element and computes the mean distance from the centroid, $dist$, and standard distance deviation σ. Features with more significant spaces than f times σ from the mean length are filtered out.

The remaining elements are then divided into layers based on their distances from the centroid, each containing elements within a specific distance range. After the layers are determined, bidirectional kNN graphs are constructed for each layer (Algorithm 2 line $4-15$) and stored in the *layerGraphList*. During the graph construction for a layer, each element is added to the graph by using Algorithm 3, which performs a greedy search (Algorithm 4) to find the candidate neighbors of the element.

Algorithm 3: ADD$(clg, elem, m, cand)$

```
Input: current layer graph clg, element to add elem, graph degree m, size of dynamic candidate list cand
Output: Update clg inserting element elem
1  $D \leftarrow \phi$ //current nearest neighbor set;
2  $m_{max} \leftarrow 2 \times m$ ;
3  $ep \leftarrow$ get the enter point of $clg$;
4  $D \leftarrow$ GET-NEAREST$(elem, ep, cand)$;
5  $neighbors \leftarrow$ add bi-directional connections from $D$ to $q$;
6  for each $n \in neighbors$ do
7      $nConn \leftarrow neighborhood(n)$ ;
8      if $|nConn| > m_{max}$ then
9          $nNewConn \leftarrow$ remove farthest element from $nconn$;
10         set $neighborhood(n)$ to $nNewConn$ ;
11     end
12     ep $\leftarrow D$;
13 end
```

Algorithm 4: GET-NEAREST$(q, ep, cand)$

```
Input: query element $q$, entry point $ep$, size of dynamic candidate list $cand$
Output: $cand$ closest neighbors to $q$
1  $V \leftarrow ep$ //visited elements set;
2  $C \leftarrow ep$ //candidate element set;
3  $D \leftarrow ep$ // list of nearest neighbors;
4  while $C>0$ do
5      $c_{min} \leftarrow$ extract nearest neighbor from $C$ to $q$ ;
6      $c_{max} \leftarrow$ extract farthest neighbor from $D$ to $q$;
7      if $distance(c_{min}, q) > distance(c_{max}, q)$ then
8          break;
9      end
10     foreach $c \in neighborhood(c_{min})$ do
11         if $c \notin V$ then
12             $V \leftarrow V \cup c$ ;
13             $c_{max} \leftarrow$ extract farthest neighbor from $D$ to $q$;
14             if $distance(c, q) < distance(c_{max}, q) or |D| < cand$ then
15                 $C \leftarrow C \cup c \quad D \leftarrow D \cup c$ ;
16                 if $|D| > cand$ then
17                     remove farthest element from $D$ to $q$;
18                 end
19             end
20         end
21     end
22 end
23 return $D$ ;
```

The algorithm starts building the bidirectional kNN graphs from the outermost layer. All the neighboring connections m are within the same layer for the outermost layer since there is no next layer to the outer layer. For the second to the outermost layer, each feature vector has $m-1$ neighboring connections within the same layer and 1 neighboring connection with the next-layer feature vector, maintaining a total of m connections for each feature vector. Next, we step towards the innermost layer, the in-layer links decrease (Algorithm 2 line 16), and the next-layer connections increase (Algorithm 2 line $8-13$), maintaining a consistent graph degree of m for all the feature vectors in dataset P. Algorithm 2 line 9-12 ensures the next-layer connections, which helps to capture the global structure of the data. New insertion of neighbors in the graph (Algorithm 2 line 10) is simply an SGR search (Algorithm 5) in the existing index. The graph is constructed by taking a simple union of all the graphs for each layer (Algorithm 2 line 18). The stratified graph captures the local and global structure of the data and can be used for efficient similarity searches. Algorithm 4 describes retrieving the candidate nearest neighbors to an element. The dynamic list D, which consists of the *cand* closest parts found, is maintained during the search. Initially, this list is populated with entry points. As the search progresses, this list is continuously updated by exploring the neighborhood of the closest element that has not been evaluated previously in the list. The process continues until the neighborhoods of all components within the list have been assessed.

The stratified graph indexing (SG) has two phases. In the first phase, elements are added simultaneously, and the computational complexity is $O\big(|P|log\big(|P|\big)\big)$. The second phase consists of a series of ANN searches at different layers and has a complexity of $O\big(|P|log\big(|P|\big)\big)$. Thus, the overall complexity of the index building of the stratified Graph (SG) scales as $O\big(|P|log\big(|P|\big)\big)$.

4 The Stratified Graph Retrieval SGR

The Stratified Graph Retrieval (SGR) steps are illustrated in Figure4 (center) and (right). The search within an index starts with a random feature vector at the innermost layer denoted in orange in Figure 4(center) and (right), where each feature has the highest number of next-layer neighbors. Then a greedy search with the graph is applied to retrieve the closest neighbor (denoted in red in Figure 4(center) and (right)) and top k to the query (marked in green in Figure 4(center) and (right)) are returned: for this example, k is 1. The heap of size *cand* maintains the neighbor list based on their distances to the query along the search path. The parameter *cand* also determines the search depth that can be performed graphically. The layering enhances the search speed of the SGR algorithm by skipping visiting nodes if the current node and query are some layers apart from each other.

Algorithm 5: SGR$(g, q, k, cand)$

```
Input: graph index $g$, query element $q$, number of nearest
       neighbors $k$, size of dynamic candidate list $cand$
Output: $k$ closest neighbors to $q$
1 $ep \leftarrow$ get entry point of $g$ ;
2 $D \leftarrow \phi$ // current nearest neighbor set;
3 $D \leftarrow GET-NEAREST(q, ep, cand)$ ;
4 $p \leftarrow$ top $k$ closest from $C$ to $q$ ;
5 return $p$
```

Figure 4 (center) and Figure 4 (right) utilizing the next layer neighbor that is closer to the query in the feature space. In other cases, the search algorithm (Algorithm 5) will perform a simple greedy search graph to retrieve the nearest neighbors to the query. Algorithm 4 demonstrates how the greedy search process works in the Stratified Graph Retrieval (SGR) technique. The proposed algorithm first identifies the nearest neighboring point p to the incoming query q by examining the neighbor list of the starting point ep. Then, it identifies the top k neighbors from p to query q. The number of hops and the average degree of the items on the greedy path are multiplied to obtain the total number of distance calculations. The SGR approach benefits from the outer layer connections that enable it to bypass visiting a considerable portion of the graph, leading to logarithmic time complexity. Thus, the time complexity of the SGR technique can be expressed as $O\big(log(|P|)\big)$.

5 Proof Of Concept

In this section, we describe how the Stratified Graph (SG) method is compared to six different state-of-the-art methods: Lightweight approximate Nearest-Neighbor library (N2) [38], Non-Metric Space Library (NMSlib) [39], Hierarchical Navigable Small World library (HNSWlib) [28], Facebook AI Similarity Search library (FaissHNSW) [40], Approximate Nearest-Neighbors Oh Yeah (Annoy) [23], and Hybrid Approximate Nearest-Neighbor Indexing and Search (HANNIS) [16].

Table 2: Dataset characteristics

Dataset	Descriptor Type	Feature Extractor	Dim	DB size in GB	# Instances in millions	% of 0s
VisDrone [12]	Video	[5]	1024	6.2	1.51	69
DOTA2.0 [13]	Image	[5]	1024	11.1	2.69	80
DIOR [14]	Image	[5]	1024	5.2	1.27	83
DEEP10M [8]	Image	[8]	96	3.8	10	0
SIFT10M [41]	Image	[41]	128	0.52	10	25
Crawl840B [42]	Text	[42]	300	5.6	2.2	0

Datasets: Table 2 summarizes real data used for the proof of concept. **VisDrone** dataset contains 1,515,007 instances of 1024 dimensional object deep feature descriptors extracted from VisDrone video [12]. **DOTA2.0** dataset contains 2,697,873 instances of 1024-dimensional object deep feature descriptors extracted from DOTA2.0 [13]. **DIOR** dataset contains 1,278,863 instances of 1024 dimensional object deep feature descriptors extracted from DIOR [14]. For the VisDrone, DOTA2.0, and datasets, we extracted 1024 dimensional object-level deep features from the last fully-connected layer using pipeline SOD [5]. **DEEP10M** dataset contains 10 million instances of 96-dimensional floating vectors. The original 10 million 1024-dimensional image embedding outputs of the Googlenet's last fully connected layer [8] were compressed and normalized into 96-dimensional vectors using principal component analysis. **SIFT10M** dataset contains 10,000,000 instances of 128-dimensional integer SIFT image descriptors [43] extracted from Caltech-256 $41 \prod 41$ whole image patches [41]. Crawl840B dataset with 300 dimensions and 2.2 million instances of vector embeddings of common crawl words using GloVe [42]. VisDrone, DOTA2.0, DIOR, DEEP10M, SIFT10M, and Crawl840B dataset sizes are 6.2, 11.1, 5.2, 3.8, 5.6, 0.516, and 11.1, in Giga Bytes, respectively.

Measures The task is visual searching for unknown objects or class discovery in the petabytes of image and video archives. Therefore, the Mean Average Precision (MAP@k) and Average Recall (AR@k) must remain consistent as k increases while keeping average retrieval time and index size comparable to the state-of-the-art for the indexing and search methods. We also include the word2Vec dataset and the SIFT10M datasets to illustrate the varying effects of different indexing methods based on the application. The performance measurement metrics have been used to evaluate the performance of each technique: MAP@k, AR@k, Average Retrieval time, and Index size. The size of retrieved se is k, $k \in [5, 10, 20, 50, 100]$.
Precision@k for query q, $P_{kq} = (|M_q \cap GT_q|)\ /\ |M_q|$.
Mean Average Precision (AP_{kq}) is calculated for each query q by considering the Precision at each rank position where relevant items are found in depth k: $AP_{kq} = \sum[P_{kq} \times rel_{kq}]/|GT_q|$. The rel_{kq} is a binary indicator equal to 1 if the item at rank k is relevant and 0 otherwise. **MAP@k** is calculated by averaging the AP values across all queries in the set Q: MAP@k$= \frac{\sum_{q \in Q} AP_k q}{|Q|}$
Recall@k (R@k) for each query q is defined as: $R_{kq} = (|M_q \cap GT_q|)\ /\ |M_q|$.
AR@k is a measure of the ability of a retrieval system to retrieve all relevant items from the entire collection across all queries in the set Q. Averaging the R_{kq} values for all queries in the set Q produces AR@k: AR@k$= \frac{1}{|Q|} * \sum_{q \in Q} R_{kq}$.
Index size defines the memory cost to save the indexes in the memory.

Setup All experiments were carried out on an Ubuntu 20.04.3 server with 11th generation Intel® CoreTM i9-11900K @ 3.5GHzX16 CPU with 128GB RAM and NVIDIA GeForce RTX 3070 8GB mem GPU. The Python implementation of the SG library can be found in [44].

6 Improving Retrieval Effectiveness

Table 3: MAP and AR comparison for deep descriptors on 100 queries.

			MAP@k						**AR@k**				
Dataset	**Method**	**k**	**5**	**10**	**20**	**50**	**100**	**k**	**5**	**10**	**20**	**50**	**100**
Vis Drone [12]	N2		0.24	0.22	0.14	0.09	0.04		0.56	0.56	0.61	0.66	0.69
	NMSlib		0.34	0.18	0.09	0.04	0.02		0.48	0.41	0.38	0.32	0.32
	HNSWlib		0.76	0.54	0.50	0.49	**0.38**		0.86	0.79	0.80	0.88	**0.91**
	FaissHNSW		0.58	0.44	0.42	0.18	0.09		0.72	0.68	0.66	0.60	0.48
	Annoy		0.40	0.27	0.21	0.19	0.13		0.50	0.50	0.53	0.66	0.73
	HANNIS		0.66	0.55	0.43	0.32	0.26		0.74	0.75	0.74	0.80	0.80
	SG		**0.80**	**0.79**	**0.74**	**0.61**	0.36		**0.96**	**0.95**	**0.96**	**0.94**	**0.91**
DOTA 2.0 [13]	N2		0.80	0.88	0.83	**0.80**	0.66		0.86	0.97	0.97	**0.97**	**0.97**
	NMSlib		0.28	0.22	0.13	0.05	0.03		0.58	0.71	0.67	0.62	0.60
	HNSWlib		0.96	0.79	0.66	0.65	0.44		0.98	0.93	0.95	0.95	0.95
	FaissHNSW		0.71	0.60	0.46	0.20	0.09		0.84	0.83	0.78	0.74	0.62
	Annoy		0.69	0.48	0.32	0.39	0.33		0.86	0.78	0.75	0.82	0.85
	HANNIS		0.94	0.88	0.83	0.75	**0.70**		0.98	0.98	0.98	0.96	0.96
	SG		**1**	**1**	**0.98**	**0**.72	0.51		**1**	**1**	**0.99**	**0.97**	0.95
DIOR [14]	N2		**1**	0.85	0.78	0.73	0.71		**1**	0.97	0.98	0.99	**0.99**
	NMSlib		0.27	0.18	0.11	0.05	0.02		0.66	0.74	0.76	0.73	0.73
	HNSWlib		0.93	0.91	0.95	0.89	0.90		0.96	0.98	0.99	0.99	0.99
	FaissHNSW		0.9	0.9	0.79	0.37	0.16		0.9	0.9	0.87	0.86	0.72
	Annoy		0.71	0.63	0.62	0.52	0.54		0.88	0.87	0.89	0.92	0.94
	HANNIS		**1**	**1**	**1**	0.98	0.90		**1**	**1**	**1**	**1**	**0.99**
	SG		**1**	**1**	**1**	**1**	**0.93**		**1**	**1**	**1**	**1**	**0.99**

The stratified Graph approach is compared with six existing methods for 2.7 million DOTA 2.0, 1.3 DIOR, and 1.5 million VisDrone 1024 dimensional vectors created using [5, 45]. Table 3 shows that Stratified Graph (SG) is the most suitable algorithm to match deep features and thus uncover similar unlabeled objects for the DOTA2.0, DIOR, and VisDrone datasets in terms of MAP and AR. Methods N2 and HNSWlib perform on par with Stratified Graph (SG) and in terms of effectiveness *only* for larger retrieval sets as illustrated in Table 3. FaissHNSW and Annoy performance quickly degrades for $k>5$, so the methods are unsuitable for deep descriptor database matching. The NMSlib consistently has low MAP and AR for all k in Table 3.

Table 4: MAP and AR for standard descriptor databases on 100 queries.

Dataset	**Method**	k	**MAP@k**					k	**AR@k**				
			5	**10**	**20**	**50**	**100**		**5**	**10**	**20**	**50**	**100**
DEEP 10M [8]	N2		0.30	0.30	0.30	0.21	0.27		0.62	0.75	0.79	**0.84**	**0.87**
	NMSlib		**0.96**	**0.81**	0.60	0.25	0.12		**0.98**	**0.95**	**0.90**	0.76	0.65
	HNSWlib		0.70	0.56	0.46	0.33	0.30		0.80	0.75	0.77	0.80	0.85
	FaissHNSW		0.69	0.45	0.29	0.11	0.05		0.78	0.69	0.60	0.56	0.45
	Annoy		0.30	0.23	0.13	0.22	0.19		0.54	0.58	0.64	0.73	0.78
	HANNIS		0.76	0.61	0.49	0.38	0.34		0.86	0.80	0.79	0.80	**0.87**
	SG		0.78	0.72	**0.63**	**0.52**	**0.41**		0.92	0.88	0.87	**0.84**	0.84
SIFT 10M [41]	N2		0.56	0.47	0.36	0.24	0.18		0.78	0.82	0.82	0.86	0.87
	NMSlib		0.22	0.12	0.06	0.03	0.01		0.64	0.66	0.63	0.58	0.52
	HNSWlib		0.85	0.76	0.52	0.27	0.25		0.94	0.91	0.84	0.82	0.84
	FaissHNSW		0.71	0.43	0.24	0.08	0.04		0.86	0.79	0.72	0.58	0.46
	Annoy		0.13	0.21	0.16	0.09	0.11		0.42	0.44	0.50	0.60	0.71
	HANNIS		0.86	0.74	0.63	**0.42**	0.22		0.94	0.93	0.93	**0.92**	**0.90**
	SG		**0.94**	**0.83**	**0.74**	**0.42**	**0.27**		**0.96**	**0.98**	**0.96**	0.90	0.82
Crawl 840B [42]	N2		0.61	0.65	0.65	0.65	0.48		0.66	0.78	0.91	0.95	**0.97**
	NMSlib		0.55	0.40	0.26	0.11	0.05		0.78	0.78	0.72	0.64	0.57
	HNSWlib		0.2	0.15	0.16	0.18	0.25		0.2	0.25	0.36	0.53	0.67
	FaissHNSW		0.3	0.25	0.16	0.06	0.03		0.42	0.45	0.45	0.38	0.28
	Annoy		0.53	0.50	0.39	0.35	0.34		0.76	0.76	0.76	0.76	0.74
	HANNIS		**0.94**	**0.92**	**0.91**	**0.83**	**0.76**		**0.98**	**0.97**	**0.98**	**0.96**	0.96
	SG		0.70	0.64	0.57	0.41	0.31		0.70	0.70	0.70	0.69	0.70

DEEP10M dataset contains 10 million instances of 96-dimensional floating vectors. The original 10 million 1024-dimensional image embedding outputs of the Googlenet's last fully connected layer [8] were compressed and normalized into 96-dimensional vectors using principal component analysis. The AP@k and AR@k-retrievals in Table 4 demonstrate that the Stratified Graph (SG) performs well in the retrieval effectiveness at higher k. The three best competitors of SG are N2, NMSlib, and HANNIS, which show inconsistent retrieval performance in Table 4. We also observe an interesting behavior of N2 and Annoy in Table 4: the effectiveness of the indexing method is *improving* with larger k. The HNSWlib performs moderately, and FaissHNSW consistently performs poorly in Table 4 compared to SG. Though DEEP10M features were extracted from DNN, the compression with principle component analysis alters the original characteristics of the dataset. The Stratified Graph (SG) is still the most consistent and suitable algorithm for discovering unknown classes for the DEEP10M dataset.

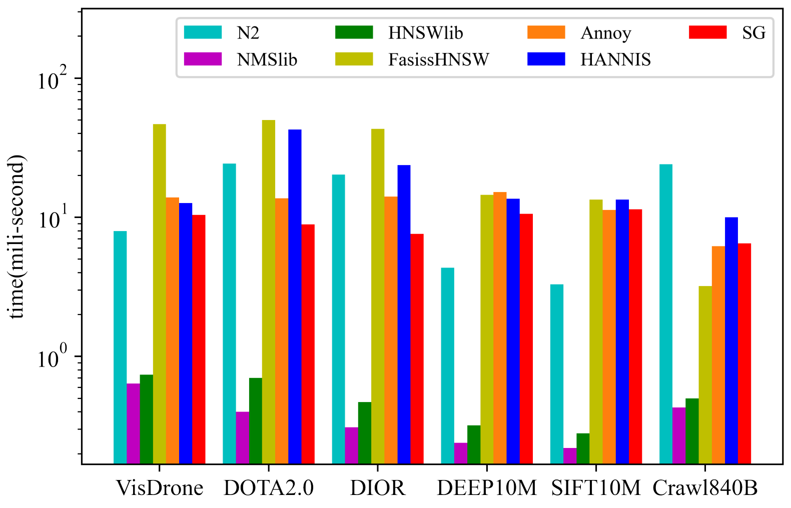

Fig. 5: Average retrieval time for 100 nearest neighbor searches.

Table 5: Index size (GB) per dataset.

Method Dataset	N2	NMS lib	HNSW lib	Faiss HNSW	Annoy	HANNIS	SG
DEEP10M	4.3	5.4	5.3	5.3	*12.8*	8.7	**2.3**
SIFT10M	2.2	2.7	2.6	2.6	*5.6*	3.9	**1.1**
Crawl840B	**2.7**	3	3	3	*4.6*	3.7	2.8
DOTA2.0	11.2	11.5	11.5	11.4	*14.2*	12.3	**3.2**
DIOR	5.3	5.3	5.4	5.4	*6.8*	5.8	**1.4**
VisDrone	6.2	6.2	6.4	6.4	*7.6*	6.9	**2.5**

Table 4 shows the MAP@k and AR@k retrieval results for the SIFT10M dataset for SG and six comparing methods. SG offers the dominating performance in MAP@k retrieval results at **all** $k \in [5, 10, 20, 50, 100]$ over the comparing methods. HANNIS is shown to be the best competitor of SG in AR@k for higher retrieval results in Table 4. N2 and Annoy show an upward trend in AR@k retrieval with larger values of k in Table 4. FaissHNSW and NMSlib offer consistently lower performance than SG for both MAP@k and AR@k retrieval results in Table 4. HNSWlib performs well and achieves similar MAP@k and AR@k for higher retrieval results. The MAP@k and AR@k retrieval results for the Crawl840B dataset are shown in Table 4. Although the algorithm is specifically designed for deep descriptor database, the performance of SG in Table 4 is compatible with the comparing methods for the vector representation of word embeddings data Crawl840B. Even though SG is designed explicitly for similarity search over deep descriptors, its performance is compatible with state-of-the-art algorithms for other descriptor databases.

7 Boosting Indexing Efficiency

In this experiment, we analyze the seven methods' retrieval time on a logarithmic scale for the six datasets *per method*. The Stratified Graph (SG) library is written in Python without any optimization for speed. The retrieval times for SG in Figure 5 shows the promising result on 100 nearest neighbor search. Stratified Graph Retrieval (SGR) is faster than FaissHNSW, Annoy, and HANNIS library for all six datasets except Crawl840B in Figure 5. The SGR is faster than N2 for DOTA2.0, DIOR, and Crawl840B datasets. The NMSlib and HNSWlib are faster than SGR for all six datasets. The retrieval time per method (5) for $k = 100$ shows N2, NMSlib, HNSWlib, FaissHNSW, and HANNIS, the retrieval time corresponds to the dimension of the dataset except for the Crawl840B dataset. Annoy has moderate retrieval time and does not correlate with the dataset size in instances and the extent and type of feature. The stratified Graph (SG) seems to do well with larger datasets. Table 5 shows the memory cost of saving the indexes in the memory. SG has the smallest index sizes compared to the comparing methods for all six datasets other than Crawl840B. Here, we use 75 trees for Annoy and 16 neighbor connections for the other six methods. NMSlib, HNSWlib, and FaissHNSW have similar index sizes in the memory. Annoy and HANNIS have larger index sizes than all the comparing methods. N2, NMSlib, HNSWlib, FaissHNSW, and HANNIS algorithms are built on the HNSW algorithm. HNSW arranges the feature vectors in a hierarchical layer of proximity graphs where the upper layers in the hierarchy are subsets of the lower layer. Therefore, the graph index containing the proximity graphs has to store way more edge lists than SG, resulting in a higher memory overhead. Annoy has the most significant index size of all the comparing methods because it requires storing many trees for better performance. SG requires up to four times less memory than comparing methods.

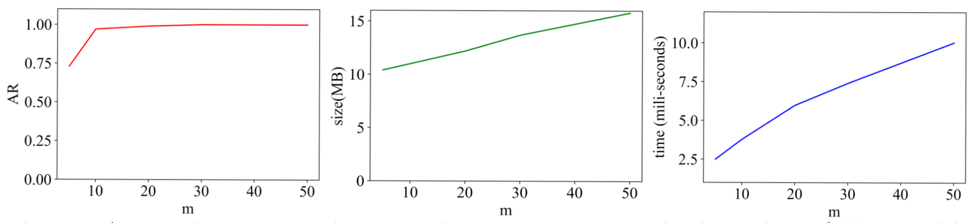

Fig. 6: AR, index size, and retrieval time increase with the value of the neighbor connection m.

8 Ablation Study

The neighbor connection m value ranges from 5 to 48 during index building. The exact value depends on the characteristics of the deep descriptor databases. The ablation study for a random dataset of dimension x and size y is illustrated in Figure 6. The AR increases and approaches 1 as we increase the number of neighbor connections m from 5 to 48 in Figure 6. We achieve an AR of 1 for this random dataset at around $m = 16$. Index size and retrieval time also increase with the value of m. Therefore, the trade-off between effectiveness and efficiency depends on the number of neighbors connected m.

9 Conclusion

We propose Stratified Graph (SG) indexing and search as an effective solution for deep-descriptor matching in large, diverse databases defined for multiple real sets. The proposed SG method outperforms state-of-the-art indexing and searching approaches in AR and MAP at the cost of slightly higher retrieval times. The MAP and AR improve up to 8% at depth 100 for the deep feature databases. SG reduces the memory cost up to **four** times compared to the state-of-the-art. Next, we will optimize the Stratified Graph (SG) method for efficient deep descriptor matching in billion deep descriptor databases.

Declarations

Authors' Contribution: Rahman took the lead on running all the experiments and prepared data and figures, and Tešić took the lead on writing, editing, and submitting the manuscript.
Compliance with Ethical Standards: The authors are complying with the ethical standards as defined by Springer. There is no potential conflict of interest, and no research involving human participants was conducted for this paper.
Competing Interests: The authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.
Research Data Policy and Data Availability: The datasets used in Sec. 5-Sec. 8 are publicly available datasets, and the download links are listed in Table 2. The code used in the paper is published here: [44].

References

[1] Heyse, D.B., Warren, N., Tešić, J.: Identifying maritime vessels at multiple levels of descriptions using deep features. In: Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications, vol. 11006, pp. 423–431 (2019). SPIE

[2] Biswas, D., Tešić, J.: Small object difficulty modeling for objects detection in satellite images. In: IEEE 14th International Conference on Computational Intelligence and Communication Networks (CICN), pp. 125–130 (2022). https://doi.org/10.1109/CICN56167.2022.10008383

[3] Zhou, X., Koltun, V., Krähenbühl, P.: Probabilistic two-stage detection. arXiv preprint arXiv:2103.07461 (2021)

[4] Bochkovskiy, A., Wang, C.-Y., Liao, H.-Y.M.: Yolov4: Optimal speed and accuracy of object detection. arXiv preprint arXiv:2004.10934 (2020)

[5] D. Biswas, Tešić, J.: Domain adaptation with contrastive learning for object detection in satellite imagery. IEEE Transactions on Geoscience and Remote Sensing (**under review**) (2023)

[6] Zhu, X., Lyu, S., Wang, X., Zhao, Q.: Tph-yolov5: Improved yolov5 based on transformer prediction head for object detection on drone-captured scenarios. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 2778–2788 (2021)

[7] Rahman, M., Tešić, J.: Hybrid approximate nearest neighbor indexing and search (hannis) for large descriptor databases. In: 2022 IEEE International Conference on Big Data, pp. 3895–3902 (2022). https://doi.org/10.1109/BigData55660.2022.10020464

[8] Baranchuk, D., Babenko, A., Malkov, Y.: Revisiting the inverted indices for billion-scale approximate nearest neighbors. CoRR **abs/1802.02422** (2018) 1802.02422

[9] Rublee, E., Rabaud, V., Konolige, K., Bradski, G.: Orb: An efficient alternative to sift or surf. In: 2011 International Conference on Computer Vision, pp. 2564–2571 (2011). https://doi.org/10.1109/ICCV.2011.6126544

[10] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)

[11] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., *et al.*: Searching for mobilenetv3. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 1314–1324 (2019)

[12] Zhu, P., Wen, L., Bian, X., Ling, H., Hu, Q.: Vision meets drones: A challenge. arXiv preprint arXiv:1804.07437 (2018)

[13] Xia, G.-S., Bai, X., Ding, J., Zhu, Z., Belongie, S., Luo, J., Datcu, M., Pelillo, M., Zhang, L.: Dota: A large-scale dataset for object detection in aerial images. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3974–3983 (2018)

[14] Li, K., Wan, G., Cheng, G., Meng, L., Han, J.: Object detection in optical remote sensing images: A survey and a new benchmark. ISPRS Journal of Photogrammetry and Remote Sensing **159**, 296–307 (2020)

[15] Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. Journal of the ACM (JACM) **45**(6), 891–923 (1998)

[16] Rahman, M., Tešić, J.: Evaluating hybrid approximate nearest neighbor indexing and search (hannis) for high-dimensional image feature search. In: 2022 IEEE International Conference on Big Data (Big Data), pp. 6802–6804 (2022). https://doi.org/10.1109/BigData55660.2022.10021048

[17] Fu, C., Wang, C., Cai, D.: High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. IEEE Transactions on Pattern Analysis and Machine Intelligence (2021)

[18] Zhang, J., Ma, R., Song, T., Hua, Y., Xue, Z., Guan, C., Guan, H.: Hierarchical satellite system graph for approximate nearest neighbor search on big data. ACM/IMS Transactions on Data Science (TDS) **2**(4), 1–15 (2022)

[19] Ono, N., Matsui, Y.: Relative nn-descent: A fast index construction for graph-based approximate nearest neighbor search. In: Proceedings of the 31st ACM International Conference on Multimedia. MM '23, pp. 1659–1667. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3581783.3612290 . https://doi.org/10.1145/3581783.3612290

[20] Zheng, B., Xi, Z., Weng, L., Hung, N.Q.V., Liu, H., Jensen, C.S.: Pm-lsh: A fast and accurate lsh framework for high-dimensional approximate nn search. Proceedings of the VLDB Endowment **13**(5), 643–655 (2020)

[21] Li, M., Zhang, Y., Sun, Y., Wang, W., Tsang, I.W., Lin, X.: I/o efficient approximate nearest neighbour search based on learned functions. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 289–300 (2020). https://doi.org/10.1109/ICDE48307.2020.00032

[22] Kim, S., Yang, H., Kim, M.: Boosted locality sensitive hashing: Discriminative binary codes for source separation. In: ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 106–110 (2020). IEEE

[23] Bernhardsson, E.: Annoy: Approximate Nearest Neighbors in C++/Python. (2018). Python package version 1.17.1. https://pypi.org/project/annoy/

[24] Gallego, A.J., Rico-Juan, J.R., Valero-Mas, J.J.: Efficient k-nearest neighbor search based on clustering and adaptive k values. Pattern recognition **122**, 108356 (2022)

[25] Jegou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. IEEE transactions on pattern analysis and machine intelligence **33**(1), 117–128 (2010)

[26] Li, W., Zhang, Y., Sun, Y., Wang, W., Li, M., Zhang, W., Lin, X.: Approximate nearest neighbor search on high dimensional data: experiments, analyses, and improvement. IEEE Transactions on Knowledge and Data Engineering **32**(8), 1475–1488 (2019)

[27] Yihe, D., Piotr, I., Ilya, R., Tal, W.: Learning space partitions for nearest neighbor search. Bulletin of the Technical Committee on Data Engineering **47**(3), 55–68 (2023)

[28] Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE transactions on pattern analysis and machine intelligence **42**(4), 824–836 (2018)

[29] Fu, C., Xiang, C., Wang, C., Cai, D.: Fast approximate nearest neighbor search with the navigating spreading-out graph. Proc. VLDB Endow. **12**(5), 461–474 (2019) https://doi.org/10.14778/3303753.3303754

[30] Beygelzimer, A., Kakade, S., Langford, J.: Cover trees for nearest neighbor. In: Proceedings of the 23rd International Conference on Machine Learning, pp. 97–104 (2006)

[31] Fu, C., Cai, D.: Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. arXiv preprint arXiv:1609.07228 (2016)

[32] Fan, X., Wang, X., Lu, K., Xue, L., Zhao, J.: Tree-based search graph for approximate nearest neighbor search. arXiv preprint arXiv:2201.03237 (2022)

[33] Iwasaki, M.: Ngt: Neighborhood graph and tree for indexing (2015)

[34] Iwasaki, M., Miyazaki, D.: Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. CoRR **abs/1810.07355** (2018) 1810.07355

[35] Iwasaki, M.: Pruned bi-directed k-nearest neighbor graph for proximity search. In: International Conference on Similarity Search and Applications, pp. 20–33 (2016). Springer

[36] Wang, Y., Ma, H., Wang, D.Z.: Lider: An efficient high-dimensional learned index for large-scale dense passage retrieval. Proc. VLDB Endow. **16**(2), 154–166 (2022) https://doi.org/10.14778/3565816.3565819

[37] Minjia, Z., Jie, R., Zhen, P., Ruoming, J., Dong, L., Bin, R.: iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures. Bulletin of the Technical Committee on Data Engineering **47**(3), 22–38 (2023)

[38] Lee, G.: TOROS N2 - Lightweight Approximate Nearest Neighbor Library Which Runs Fast Even with Large Datasets. (2017). Python package version 0.1.7. https://github.com/kakao/n2

[39] Boytsov, L., Naidan, B.: Engineering efficient and effective non-metric space library. In: International Conference on Similarity Search and Applications, pp. 280–293 (2013). Springer

[40] Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with GPUs. IEEE Transactions on Big Data **7**(3), 535–547 (2019)

[41] Dua, D., Graff, C.: UCI Machine Learning Repository (2017). http://archive.ics.uci.edu/ml

[42] Pennington, J., Socher, R., Manning, C.D.: Glove: Global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1532–1543 (2014)

[43] Lowe, G.: Sift-the scale-invariant feature transform. Int. J **2**(91-110), 2 (2004)

[44] Rahman, M., Tešić, J.: Stratified Graph Indexing for Efficient Search in Deep Descriptor Databases. (2023). https://anonymous.4open.science/r/SG-4644

[45] Biswas, D., Tešić, J.: Small Object Detection Feature Extractor. (2023). https://github.com/DataLab12/SOD