Stratified Graph Indexing for Efficient Search in Deep Descriptor Databases

M M Mahabubur Rahman[1*] and Jelena Tešić[2]

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

*Corresponding author(s). E-mail(s): toufik@txstate.edu;
Contributing authors: jtesic@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 **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 contribute to enhancing recall, 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 Stratified Graph (SG) indexing and search has smaller memory usage (up to four times), and farther precision and recall improves up to 8% than state-of-art for all six datasets at five retrieval depths.

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

1 Introduction

The world we live in today is a treasure trove of video archives that houses an immense amount of valuable information. To tap into the true potential of these archives, it's crucial to identify objects that are similar to one another as object labeling is sparse and far apart. Achieving this task requires a similarity search in deep descriptors databases that can detect comparable objects that are widely spread throughout the dataset. In this paper we propose to answer the task of *finding the appearances of the new unlabeled object in the existing video archives?*, formulate the task at hand as as the *k-Nearest Neighbor task in the deep descriptor space.*

Deep features extracted from deep neural networks have been proven to capture the object representation well, no matter how small objects are or how diverse the dataset is [1], and recent object detectors built on CenterNet2[2], YOLOv4 [3], SOD [4], and TPH-YOLOv5 [5] provide an efficient 1D representation of 2D objects in a video. Similarity search looks for object representations in a database that are similar or close to a query based on a specific measure of similarity. 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. This approach does not scale for $|P|$ in millions or billions and d in hundreds and thousands as the k-NN search considers the entire dataset each time a query is initiated. Our original task of finding a needle in the haystack and searching the entire haystack every time is reformulated as an indexing and search problem. First, we construct the data structure that summarizes point dataset P with the objective of enabling efficient nearest neighbor retrieval without the need to compute all distances from query vector q to all the points in P while trying to match the full exact retrieval set as closely as possible. The approximate nearest-neighbor (ANN) methods are traditionally optimized to balance efficiency and effectiveness and offer speed up at the account of the accuracy [6]. ANN methods are optimized to return *any* point $p' \in P$ such that the distance from q to p' is at most $c \cdot min_{p \in P} D\big(q, p\big)$, for some $c \geq 1$ fast, and they can be roughly grouped as graph-based [7–9], hashing-based [10–12], and partition-based [13, 14] methods. The proposed methods can be applied to a variety of feature databases, and the effectiveness varies based on the size and application as indexing and search approaches are correlated with data characteristics. Deep descriptor databases are high dimensional (1024) and sparse (58% of the feature vectors are 0).

1.1 Motivation

The motivation behind our research stems from the inherent data sparseness found in deep descriptor databases. These databases typically exhibit high dimensionality, often exceeding 1024 dimensions, while being sparse, with an average of approximately 58% of feature vectors containing zeros. To compare these deep descriptor datasets with other non-deep descriptor datasets, we provide a summary in Table 1.

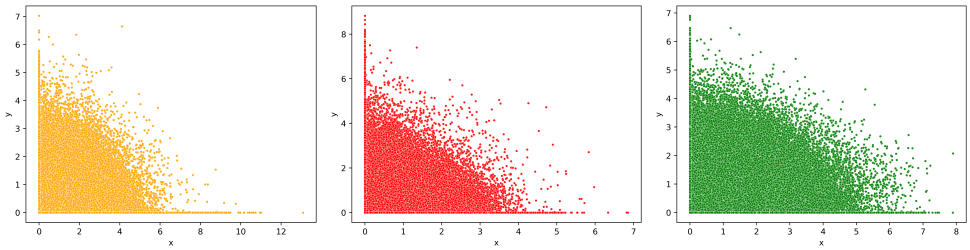


Fig. 1 Data distribution along first two dimensions for VisDrone, DOTA2.0 and DIOR.

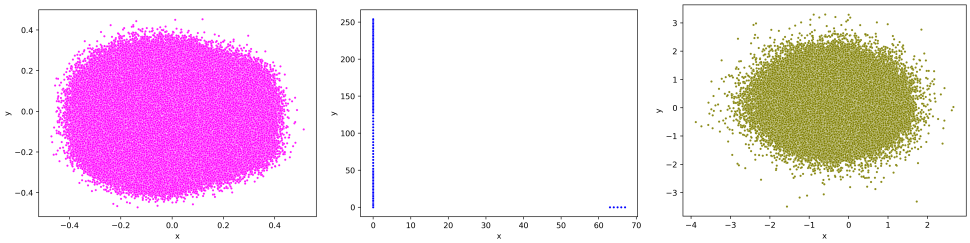


Fig. 2 Data distribution along first two dimensions for DEEP10M, SIFT10M and Crawl840B. Note that these are two first principal components of the Deep10M

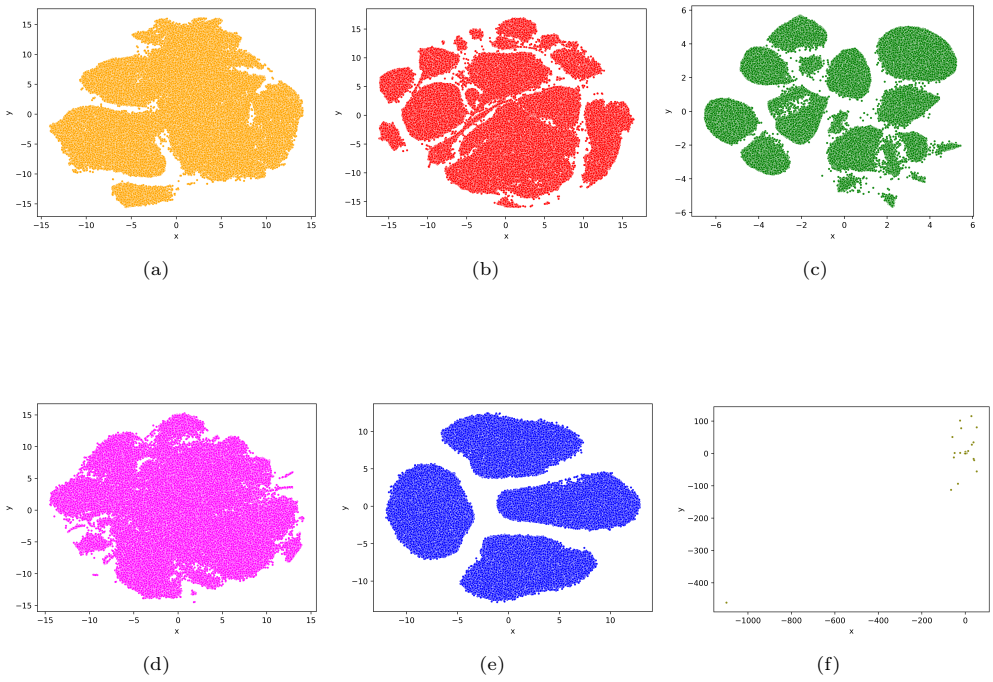
In particular, the Crawl840B dataset stands out as it contains no zero entries, while DEEP10M has only 19 entries with zeros. On the contrary, deep descriptor float databases like DOTA2.0, DIOR, and Visdrone exhibit an average of 77% zeros per feature vector, and SIFT10M contains 25% zero entries, as indicated in Table 1. This stark contrast motivated our investigation into the distribution of descriptors across dimensions and their visualization using t-SNE.

When we examine the data distributions along the first two dimensions of the entire deep float descriptor databases (VisDrone, DOTA2.0 and DIOR), as depicted in Figure 1, we observe that the data follows a similar distribution pattern to that of a DNN used to extract 1024-dimensional vectors. However, this distribution greatly differs from that of DEEP10M, SIFT10M, and Crawl840B, as shown in Figure 2. It's important to note that DEEP10M's integer feature distribution differs from other deep descriptor databases because it compresses and normalizes the 1024-dimensional deep feature vectors from Googlenet's last fully connected layer into 96-dimensional vectors using principal component analysis. Figure 2 illustrates the distribution of the first two principal components of this dataset, which is somewhat similar to the word vector distribution of Crawl840B.

Examining the data distribution along the first two dimensions of the SIFT10M image descriptor database reveals a skewed distribution, with values concentrated around a single point along each dimension. This indicates non-uniform distribution, with certain dominant patterns in the feature space, which can be attributed to the SIFT [15] algorithm's approach of creating 16 patches from an image and using gradient orientation along 8 directions.

Further insights come from the t-SNE visualizations. In Figure 3(a,b,c), we present t-SNE distributions of random 100,000 vectors from three float databases (Visdrone, DOTA2.0 and DIOR), while Figure 3(d,e,f) displays the t-SNE distribution of a sample of integer descriptor databases (DEEP10M, SIFT10M, and Crawl840B). Notably, the t-SNE visualization of DEEP10M's first 96 principal component vectors (Figure 3(d)) bears similarity to the overall mapping of the deep descriptors seen in Figure 3(a,b,c). However, SIFT descriptors cluster distinctly into four clusters (Figure 3(e)). In contrast, word vector visualization encounters challenges, with most points overlapping in a few dots in the top right corner (Figure 3(f)).

In summary, our investigation reveals a marked difference in the behavior of deep descriptor databases compared to compressed deep descriptors, traditional image descriptors, and word vector databases. 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 suited to this task.



(a) (b) (c)

(d) (e) (f)

Fig. 3 t-SNE distribution for 100,000 features randomly sampled from (a) VisDrone, (b) DOTA 2.0, (c) DIOR, (d) DEEP10M, (e) SIFT10M, and (f) Crawl840B datasets. The non-deep descriptor distribution differs in terms of cluster separation.

1.2 Proposed Work

Our task focuses on effective and scalable indexing and retrieval improvements for real deep feature databases for unknown class discovery applications. We introduce the

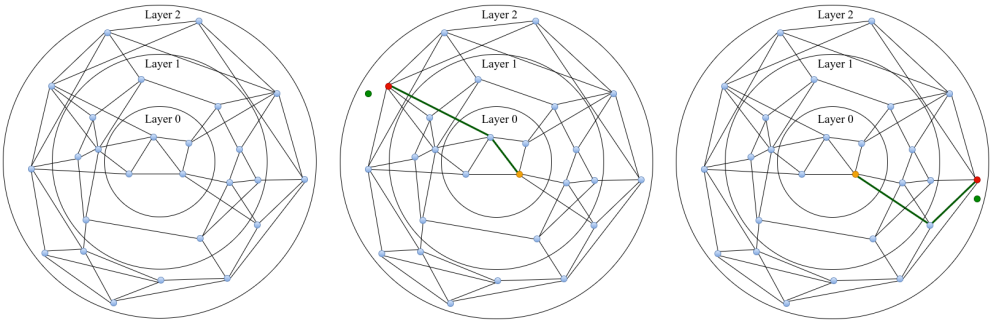
Stratified Graph (SG) as a solution for effective indexing and retrieval. The *Stratified Graph (SG)* approach is tailored to address the challenges of searching through high-dimensional and sparse datasets of deep descriptors by considering the sparsity of feature vectors. The Stratified Graph Indexing technique involves segmenting the data into layers. Each layer's composition is determined by the distance of data points from the dataset's center of mass. This means that regardless of how sparsely feature vectors are distributed within the feature space, the layers are consistently centered around the dataset's center of mass.

In each layer, the index is constructed as a bi-directional graph that links each vector to its closest neighbors within the same layer, and to its nearest neighbor in the adjacent outer layers. This layer graph is designed to achieve skip list properties which helps the searching algorithm to skip visiting nodes along the greedy search path [16]. During the search process, the Stratified Graph search starts from the innermost layer and gradually moves outwards to the outer layers through the connected graph. The rest of the paper is organized as follows. In Section 2 we related work and describe the limitations of existing methods in terms of scalability and efficiency for larger datasets. In Section 3, we discuss the indexing and search procedure in detail. 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, while the SG connections between layers help maintain logarithmic complexity scaling for faster search speed. In Section 4, we present an in-depth comparison of multiple state-of-the-art methods in 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 recall, precision, and F1-score at any depth of retrieval and fast retrieval times, and index size. We present our findings that the Stratified Graph approach is the most suitable for deep descriptor application in Section 5.

2 Related Work

The *Faiss* library[17] enables efficient partitioning of data in Voronoi cells [17], where the index of each cell is a centroid of that cell and product quantization is used [18] to compress data. The approach and its hierarchical improvements have not been shown to scale well for further retrieval results in 128 dimensional SIFT10M dataset with 10 million instances and 96 dimensional DEEP10M dataset with 10 million instances shown in Section 4. Annoy generates a number of hierarchical 2-means trees by recursive partitioning. Each iteration results in the formation of 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 of them. The hyperplane then partitions the data points into two sub-trees, and the algorithm iteratively generates the index on each sub-tree [13]. This approach did not scale to larger high dimensional datasets in terms of the speed of recovery and accuracy of the retrieved results [19]. *Hierarchical Navigable Small World* (HNSW) [20] arranges the graph into a hierarchy of proximity graph layers with lower layer containing all the feature vectors and higher layers containing a subset of previous layers in the

hierarchy. However, this architecture results in larger index size shown in Section 4. *Navigating Spread-out Graph* (NSG) [21] favors the "Navigating Node" to make the search efficient, but the sparsity of the data space and the indexing complexity does not scale well when the dimension of the feature vector grows [8]. *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. The sparsity of NSSG can be controlled by a parameter, but the chances that the monotonic search stage fails are greater as the size and dimension of the database increases [8]. The *Tree-Based Search Graph* (TBSG) proposes to handle this problem with the probability of monotonic search success by combining the Cover Tree [22] and BKNNG (Bi-directed K-Nearest Neighbor Graph) [23] algorithms [24]. 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 total number of steps in large data [9]. 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. During the search, HSSG performs a faster coarse search on the upper layer with fewer nodes. Following the coarse finds, HSSG conducts a more precise recursive fine search in the bottom layer at the cost of the high indexing and memory overhead compared to SSG [9]. *Neighborhood Graph Tree* (NGT) [25] 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 [26] and performs a range search to obtain the nearest neighbors. A major drawback of NGT is that if the query and seed are far away from each other in the search space, then it takes many hops in between to reach the query from the seed, and thus increases the retrieval time. One way to address this problem is to transform the k nearest neighbor graph into a undirected one, and the other is to construct an undirected graph by continuously inserting elements [27]. ***L**earned **I**ndex for large-scale **DE**nse passage **R**etrieval* (LIDER)'s [28] 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 key re-scaling module. High-dimensional dense embeddings are converted into one-dimensional keys and sorted in a certain order to make quick predictions by the RMI. However, for a small number of clusters, each cluster yields a large number of feature vectors, making it more difficult for RMI to learn the distribution effectively. As a result, the quality of in-cluster retrieval degrades. On the contrary, for a large number of clusters, recall suffers. All neighboring ANN methods suffer from a long index-building time and low retrieval for large deep-descriptor databases [19]. In this paper we compare and contrast the proposed work with state-of-the-art for the large sparse descriptor retrieval.



(a) (b) (c)

Fig. 4 Illustration of SG index (a) and two different scenarios (b,c) 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).

3 Stratified Graph Methodology

3.1 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 that are closer to the centroid are stratified in closer layers while the feature vectors that are farther away are stratified in farther layers. Figure 4(a) illustrates the Stratified Graph Indexing (SGI) for the Euclidean distance and the layers are stacked into the target shape in 2D. Note that for the Manhattan distance, the layers can be illustrated as a set of rotated squares in 2D illustration. The bidirectional graph is constructed by connecting each feature vector to its m-Nearest Neighbors. In Figure 4(a), m is set to 4. We specify the number of layers as $log_2 m+1$. Therefore, our example in Figure 4 has 3 layers. We select the layer width by dividing the distance to the farthest point from the center by the layer numbers. Therefore, all the layers have the same width. Four points in Layer 0 have two connections within the central layer and one each with the outside layers. In Layer 1, ten points have three connections within Layer 1 and one with the outer layer. All the points in Layer 2 have four connections within Layer 2. 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, higher index size, slower search but higher recall as each point is connected to its true m nearest neighbors (Figure 7).

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. In this process, an outlier filtering factor f is used to filter out elements that are too far from the mean distance and ensure that outliers do not affect the layer boundaries.

Algorithm 1: SGI $(SGI, P, m, f, cand)$

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

```
graph ← φ
layerGraphList ← φ
layeredElem ← LAYERING(P, m, f)
foreach layer of layeredElem do
    clg ← φ
    foreach elem of layer do
        clg ← ADD(clg, elem, m, cand)
        if layerGraphList not empty then
            foreach g in layerGraphList do
                n ← SGR(g, elem, k = 1, cand)
                clg ← update clg inserting n to neighbor list of elem
            end
        end
        add clg to layerGraphList
    end
    m ← m − 1
end
graph ← merge all the graphs in layerGraphList
```

Algorithm 2: 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  W ← SGR(clg, elem, m, cand);
2  ep ← get the nearest element from W to elem;
3  for l_c ← min(L, l) ... 0 do
4      W ← SEARCH-LAYER(q, ep, efConstruction, l);
5      neighbors ← SELECT-NEIGHBORS(q, W, M, l_c);
6      for each e ∈ neighbors do
7          if | eComn |> M_max then
8              eNeuConn ← SELECT-NEIGHBORS(e, Com, M_mar, l_c);
9              ep ← W;
10             if I > L then
11                 set enter point for hnsw to q;
12             end
13         end
14     end
15 end
```

Algorithm 3 describes the process of determining the layers for each element. The algorithm computes the mean distance from the centroid, *dist*, and the standard deviation of distances, σ. Elements whose distances are greater than f times σ from the mean distance are filtered out. The remaining elements are then divided into layers based on their distances from the centroid, with each layer containing elements within a certain range of distances. After the layers are determined, the kNN graph is constructed for each layer (Algorithm 1 line 6). Then the algorithm adds the next-layer neighbors to all the feature vectors that are not in the outermost layer, which helps to capture the global structure of the data (Algorithm 1 line 7-12). Any new insertion of neighbors in the graph is simply an SG search (Algorithm 4) in the existing index. The final graph is constructed by taking a simple union of all the graphs for each layer (Algorithm 1 line 17). The resulting graph captures the local and global structure of the data and can be used for efficient similarity search.

The stratified graph indexing (SG) has two phases. During the first phase, each element is added one at a time by iterative insertions, which are simply a series of ANN searches at different levels. Thus, the first phase has a complexity of $O\big(|P|log\big(|P|\big)\big)$.

The second phase of stratified graph (SG) index building is also a series of ANN searches at different layers. Thus, similar to the first phase, the second phase also has a complexity of $O\big(|P|log\big(|P|\big)\big)$. Therefore, the overall complexity of the index building of the stratified graph (SG) scales as $O\big(|P|log\big(|P|\big)\big)$.

3.2 The Stratified Graph Retrieval SGR

The Stratified Graph Retrieval (SGR) steps are illustrated in Figure4 (b) and (c). The search within an index starts with a random feature vector at the innermost layer

Algorithm 3: LAYERING (X, M, f)

Input: data vector P, number of established connections m, outlier filtering factor f
Output: element list with assigned layer

$numLayer \leftarrow \lfloor \log_2 m \rfloor$
$cen \leftarrow$ mean of X
$dist \leftarrow$ distances from centroid to all data vectors
$avg \leftarrow$ mean of all distances
$\sigma \leftarrow$ standard deviation of all distances
$u_b \leftarrow avg + f \times \sigma$
$l_b \leftarrow$ smallest of $dist$
$r \leftarrow \frac{u_b - l_b}{numLayer}$
$layeredElem \leftarrow \phi$
foreach (d, p) *of* $(dist, P)$ **do**
  $l \leftarrow \lfloor \frac{d}{r} \rfloor$
  add element p to layer l in $layeredElem$
end
return $layeredElem$

Algorithm 4: SGR$(g, elem, k, cand)$

Input: graph index g, query element q, number of nearest neighbors k, size of dynamic candidate list $cand$
Output: k closest neighbors to q

$ep \leftarrow$ get entry point of g

$p \leftarrow$ extract nearest neighbor to q starting with ep

$C \leftarrow$ extract $cand$ neighbors to p from g
$neighbors \leftarrow$ top k closest from C to q
return k-NN neighbors to q

denoted in orange in Figure 4(b) and (b), where each feature has the highest number of next-layer neighbors. Then a greedy search within the graph is applied to retrieve the closest neighbor (denoted in red in Figure 4(b) and (b)) and top k to the query (denoted in green in Figure 4(b) and (b)) are returned: for this example, k is 1. A 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 depth of search that can be performed in the graph. 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. Figure 4 (b) and Figure 4 (c) utilizing the next layer neighbor that is closer to the query in the feature space. In other cases, the search algorithm will perform a simple greedy search in the 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. Initially, the algorithm locates the nearest neighboring point p to the incoming query q by examining the neighbor list of the starting point ep. It then 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. Therefore, the time complexity of the SGR technique can be expressed as $O\big(log\big(|P|\big)\big)$.

4 Experiments

We measure the performance of our Stratified Graph (SG) method as compared to the six different state-of-the-art methods: Lightweight approximate Nearest-Neighbor library (N2) [29], Non-Metric Space Library (NMSlib) [30], Hierarchical Navigable Small World library (HNSWlib) [20], Facebook AI Similarity Search library (FaissHNSW) [31], Approximate Nearest-Neighbors Oh Yeah (Annoy) [13], and Hybrid Approximate Nearest-Neighbor Indexing and Search (HANNIS) library built on the HNSW algorithm [32].

Table 1 Dataset characteristics

Data	Desc. Type	Framework	Dimension	DB size in GB	# features	% 0s
VisDrone [33]	Video	[4]	1024	6.2	1.51	69
DOTA2.0 [34]	Image	[4]	1024	11.1	2.69	80
DIOR [35]	Image	[4]	1024	5.2	1.27	83
DEEP10M [17]	Image	[17]	96	3.8	10	0
SIFT10M [36]	Image	[36]	128	0.516	10	25
Crawl840B [37]	Text	[37]	300	5.6	2.2	0

4.1 Experimental Setup

Datasets characteristics used from the experiments are summarized in Table 1. **VisDrone** dataset contains 1,515,007 instances of 1024 dimensional object deep feature descriptors extracted from VisDrone video [33]. **DOTA2.0** dataset contains 2,697,873 instances of 1024-dimensional object deep feature descriptors extracted from DOTA2.0 [34]. **DIOR** dataset contains 1,278,863 instances of 1024 dimensional object deep feature descriptors extracted from DIOR [35]. For the VisDrone, DOTA2.0,

and datasets, we extracted 1024 dimensional object-level deep features from the last fully-connected layer using pipeline SOD [4]. **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 [17] 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 [15] extracted from Caltech-256 $41 \prod 41$ whole image patches [36]. Crawl840B dataset with 300 dimensions and 2.2 million instances of vector embeddings of common crawl words using GloVe [37]. 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.

Evaluation of the indexing and search methods have been conducted from the task's perspective. Our task is focused on a visual search 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) need to remain consistent as k increases while keeping average retrieval time and index size comparable to state of the art. Note that we include the word2Vec dataset and the SIFT10M databases to illustrate the varying effect of different methods based on the application. Four performance measurement metrics have been used to evaluate the performance of each method: MAP@k, AR@k, Average Retrieval time, and Index size. The number k is the size of the retrieved set, $k \in [5, 10, 20, 50, 100]$.

MAP@k is calculated by considering Precision at each rank position (k) and then averaging these Precision values across all relevant queries. For a set of queries Q, where each query q has its respective retrieval set M_q and ground truth set GT_q, calculate Precision (P@k) for each query q. Here, GT_q is the k Nearest Neighbors set of descriptor indices recovered by the brute force search for query q.

P@k for query q, $P_{kq} = (|M_q \cap GT_q|) \ / \ |M_q|$

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:

$AP_{kq} = \sum[P_{kq} \times rel_{kq}]/|GT_q|$

Here, 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:

$MAP@\text{k}= \frac{\sum_{q \in Q} AP_k q}{|Q|}$

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. Recall (R@k) for each query q is defined as:

$R_{kq} = (|M_q \cap GT_q|) \ / \ |M_q|$

Once we have the R_{kq} values for all queries in the set Q, we calculate the AR@k as the average of these Recall values:

$AR@\text{k}= \frac{\sum_{q \in Q} R_{kq}}{|Q|}$

Average Retrieval time refers to the mean time taken to retrieve the k nearest neighbors for a single query, denoted as q, from a set of queries, referred to as Q. To calculate this metric, we add up the total retrieval time for all queries in set Q and

then divide it by the total number of queries, represented as $|Q|$. **Index size** defines the memory cost to save the indexes in the memory.

Setup All experiments were carried out on 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 SG library can be found in https://anonymous.4open.science/r/SG-4644.

4.2 Effectiveness of the Retrieval Methods

Table 2 Precision and recall comparison for VisDrone.

Metrics / Methods	Precision					Recall				
	k=5	**k=10**	**k=20**	**k=50**	**k=100**	**k=5**	**k=10**	**k=20**	**k=50**	**k=100**
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**

Table 3 Precision and recall comparison for DOTA2.0.

Metrics / Methods	Precision					Recall				
	k=5	**k=10**	**k=20**	**k=50**	**k=100**	**k=5**	**k=10**	**k=20**	**k=50**	**k=100**
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

Table 4 Precision and recall comparison for DIOR.

Metrics / Methods	Precision					Recall				
	k=5	**k=10**	**k=20**	**k=50**	**k=100**	**k=5**	**k=10**	**k=20**	**k=50**	**k=100**
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**

In this experiment, we compare the Stratified Graph approach with six existing approaches for 2.7 million DOTA 2.0, 1.3 DIOR and 1.5 million VisDrone 1024 dimensional databases created using the methodology published in [4]. Table 3, 4 and 2 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 precision and recall. 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, 4 and 2. FaissHNSW and Annoy performance quickly degrades for k>5 so the methods are not suitable for deep descriptor database matching. NMSlib has consistently low precision and low recall for all k in Table 3, 4 and 2.

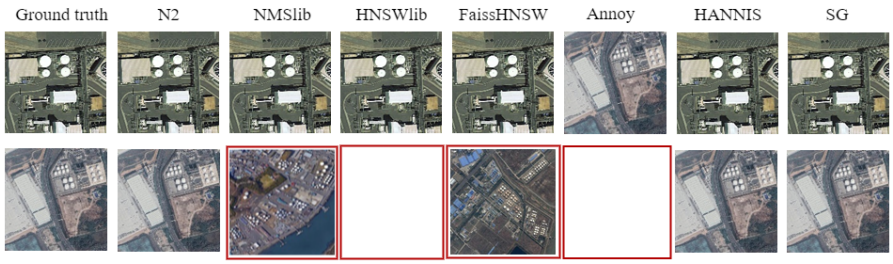


Fig. 5 Fifth and sixth retrieval results for a single query w.r.t. brute force search for seven methods N2, NMSlib, FaissHNSW, Annoy, HNSWlib, HANNIS, and Stratified Graph (SG).

Table 5 Precision and recall comparison for DEEP10M.

Metrics / Methods	Precision					Recall				
	k=5	**k=10**	**k=20**	**k=50**	**k=100**	**k=5**	**k=10**	**k=20**	**k=50**	**k=100**
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

Table 6 Precision and recall comparison for SIFT10M.

Metrics / Methods	Precision					Recall				
	k=5	**k=10**	**k=20**	**k=50**	**k=100**	**k=5**	**k=10**	**k=20**	**k=50**	**k=100**
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

Visual retrieval results for a single query for DIOR dataset are shown in Figure 5. We compare retrieved images for 10-NN search with respect to the brute force result shown as ground truth in Figure 5. All the methods return similar top 4 results as brute force results. Therefore, we only show the fifth and sixth retrieval results for all

Table 7 Precision and recall comparison for Crawl840B.

Metrics / Methods	Precision					Recall				
	k=5	**k=10**	**k=20**	**k=50**	**k=100**	**k=5**	**k=10**	**k=20**	**k=50**	**k=100**
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

the methods in 5. For this particular object deep descriptor query, N2, HANNIS and SG retrieves **all** 6 unique images correctly. Annoy retrieves 4 out of 6, and the rest retrieves 5 out of 6 images correctly. Some queries may be difficult for one method but easy for another, so we average all results throughout the experiment for 10 distinct queries.

Next, let's see if the methods behave in a similar fashion to different DNN used to extract descriptors. **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 [17] were compressed and normalized into 96-dimensional vectors using principal component analysis. Precision@k and recall@k-retrievals in Table 5 demonstrate that the Stratified Graph (SG) is performing well in the effectiveness of retrieval at higher k. The three best competitors of SG are N2, NMSlib, and HANNIS show inconsistent retrieval performance in Table 5. We also observe an interesting behavior of N2 and Annoy in Table 5: the effectiveness of the indexing method is *improving* with larger k. HNSWlib performs moderately, and FaissHNSW has consistently poor performance in Table 5 compared to SG. Our interpretation from Table 5 result is: though DEEP10M features were extracted from DNN, the compression with principle component analysis alters the original characteristics of the dataset. However, Stratified Graph (SG) is shown to be the most consistent and suitable algorithm for discovering unknown classes for the DEEP10M dataset. In summary, SG is shown to be most robust algorithm for discovering unknown classes in deep descriptor databases.

Table 6 shows the precision@k and recall@k retrieval results for the SIFT10M dataset for SG and six comparing methods. Here, SG shows the dominating performance in precision@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 recall@k for higher retrieval results in Table 6. N2 and Annoy show an upward trend in Recall@k retrieval with larger values of k in Table 6. FaissHNSW and NMSlib show consistently low performance than SG for both precision@k and recall@k retrieval results in Table 6. HNSWlib performs well and achieves similar precision@k and recall@k for higher retrieval results.

The precision@k and recall@k retrieval results for Crawl840B dataset are shown in Table 7. Though our algorithm is specifically designed for deep descriptor database, the performance of SG in Table 7 is compatible with the comparing methods for the vector representation of word embeddings data Crawl840B.

In summary, though SG is specifically designed for similarity search over deep descriptors, its performance is compatible with state-of-the-art algorithms for other descriptor databases.

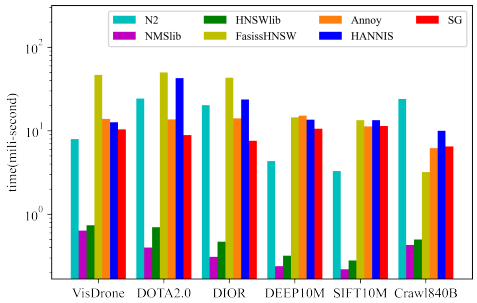


Fig. 6 Retrieval time for 100 nearest neighbor search per dataset for 7 approaches on 6 datasets.

4.3 Efficiency Comparisons of the Index Size and Retrieval Times

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. However, the retrieval times for SG in Figure 6 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 6. Moreover, SGR is faster than N2 for DOTA2.0,DIOR and Crawl840B datasets. NMSlib and HNSWlib are faster than SGR for all six datasets. The retrieval time per method (6) 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 dimension and type of feature.

Table 8 Index sizes (GB) per dataset for 7 approaches using 75 trees for Annoy and 16 neighbor connections for the rest of the six methods.

Methods/ Datasets	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**

The Stratified Graph (SG) seems to do well with larger datasets. Table 8 shows the memory cost of saving the indexes in the memory. SG has the smallest index sizes than the comparing methods for all six datasets other than Crawl840B. 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 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 largest index size of all the comparing methods because it requires storing many trees for better performance. Overall, SG requires up to four times less memory than comparing methods.

4.4 Experiment 3: Ablation Study for the number of neighbors m in SG

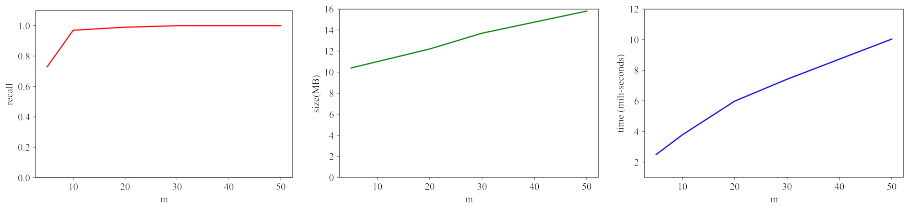


Fig. 7 Demonstration of recall, index size, and retrieval time increasing with a higher degree of m.

The suitable value of neighbor connection m during index building depends on the characteristics of the deep descriptor databases and can range from 5 to 48, and the ablation study for random dataset of dimension x and size y is illustrated in Figure 7. The recall increases and approaches 1 as we increase the number of neighbor connections m from 5 to 48 in Figure 7. For this random dataset, we achieve a recall of 1 at around $m = 16$. We see a similar trend for index size and retrieval time, both increasing with the value of m. Therefore, the trade-off between effectiveness and efficiency depends on the number of neighbors connection m. Lower value of m results in faster search loosing some accuracy.

5 Conclusion

The potential hidden within video archives is immense, but the challenge lies in the limited amount of annotated images and objects. The solution is to identify objects similar to one another in feature space, which can be accomplished through a similarity search in deep descriptors databases. 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. Furthermore, the proposed stratified graph (SG) method outperforms state-of-the-art indexing and searching approaches in terms of recall and

precision at the cost of slightly higher retrieval times. The precision, and recall improve up to 8% at depth 100 for the deep feature databases. Moreover, SG reduces the memory cost up to **four** time than comparing methods. As the next step, we plan to optimize the Stratified Graph (SG) method for efficient deep descriptor matching in billion deep descriptor databases.

References

[1] Heyse, D., NicholasWarren, Tešić, J.: Identifying maritime vessels at multiple levels of descriptions using deep features. In: Pham, T. (ed.) Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications, vol. 11006, pp. 423–431. SPIE, ??? (2019). https://doi.org/10.1117/12.2519248 . International Society for Optics and Photonics. https://doi.org/10.1117/12.2519248

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

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

[4] Biswas, D., Tešić, J.: Progressive Domain Adaptation with Contrastive Learning for Object Detection in the Satellite Imagery (2023)

[5] 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)

[6] 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)

[7] Rahman, M.M.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

[8] 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)

[9] 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)

[10] 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)

[11] 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

[12] 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

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

[14] 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)

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

[16] Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM **33**(6), 668–676 (1990)

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

[18] 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)

[19] 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)

[20] 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)

[21] Fu, C., Xiang, C., Wang, C., Cai, D.: Fast approximate nearest neighbor search with the navigating spreading-out graph. arXiv preprint arXiv:1707.00143 (2017)

[22] 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)

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

[24] 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)

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

[26] Iwasaki, M., Miyazaki, D.: Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. arXiv preprint arXiv:1810.07355 (2018)

[27] 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

[28] Wang, Y., Ma, H., Wang, D.Z.: Lider: An efficient high-dimensional learned index for large-scale dense passage retrieval. arXiv preprint arXiv:2205.00970 (2022)

[29] 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

[30] 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

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

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

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

[34] 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)

[35] 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)

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

uci.edu/ml

[37] 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)