Hybrid Approximate Nearest Neighbor Indexing and Search (HANNIS) for Large Descriptor Databases

M M Mahabubur Rahman
Computer Science
Texas State University
San Marcos,TX,USA
toufik@txstate.edu

Jelena Tešić
Computer Science
Texas State University
San Marcos,TX,USA
jtesic@txstate.edu

***Abstract*—In this paper, we present a novel method for efficient and effective retrieval of similar deep descriptors. Our new hybrid method for indexing and searching for the approximate nearest neighbors in high-dimensional large deep-descriptor databases retrieves truly similar items in the database, even if the retrieval set is large. The proposed solution —- hybrid approximate nearest neighbor indexing and search (HANNIS) —- partitions the whole data space using the kmeans++ algorithm and then indexes each cluster using adapted hierarchical navigable graphs. This approach enables us to load items that are truly close to the incoming query at retrieval time. HANNIS outperforms all state-of-the-art methods in terms of recall at depths of up to 100 and offers consistent index loading and retrieval performance.**

***Index Terms*—large set retrieval; high-dimensional indexing and search; deep descriptors; big data**

I. Introduction

The increasing amount of data requires efficient and scalable retrieval of similar instances for further analysis in most data science applications, such as data integration, recommender systems, information retrieval, software engineering, cybersecurity, outlier detection, classification, and clustering. A similarity search can be defined as a search for objects from a database that is close to a query based on some sort of similarity, also known as the distance function. According to its definition, the nearest-neighbor problem is to create a data structure that, given any point, q in a metric space (X, D), gives the point in P that is closest to q (the point *nearest neighbor* in P). The data structure maintains more details about the set P, which is then utilized to locate the closest neighbor without calculating all the distances between q and P.

The k-nearest neighbors (k-NN) search identifies the top k nearest neighbors to the query and performs very well for retrieving exact solutions in smaller data sets with a lower dimension. However, the k-NN search can be sluggish in large datasets and higher dimensions because of the "curse of dimensionality". Instead of generating a model from the training data, the k-NN search considers the entire data set each time a query is initiated. This quality leads to high memory usage and inefficiency in the retrieval result as the data increases.

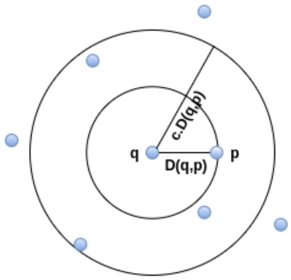

Fig. 1: ANN search

To address this problem, several approximate nearest-neighbor (ANN) methods were introduced. ANN methods work very fast but sacrifice some accuracy by loosening the condition of exact nearest-neighbor retrieval. ANN methods handle the "curse of dimensionality" well and thus have superior performance over the k-NN search in large datasets with higher dimensions. ANN methods, which aim to produce any point $p' \in P$ such that the distance from q to p' is at most $c.D(q, p)$, for some $c \geq 1$ (Fig. 1), can be used to speed up computations with closest neighbors and reduce memory usage by the data structure. Many effective ANN solutions have been proposed in the past decade and can be grouped as graph-based [1]–[5], hashing-based [6]–[8], and partition-based [9]–[11] methods. In this paper, we focus on indexing and retrieval improvements for real deep feature databases for the unknown class discovery application, and we measure algorithms in terms of high recall at any depth of retrieval, fast index loading, and retrieval times.

Some libraries [1], [9], [12], [13] have used the graph-based Hierarchical Navigable Small World (HNSW) algorithm. Among all ANN methods, the HNSW algorithm [1] shows promising results in terms of retrieval time [14]. However, there is still room for improvement in recall. Moreover, all of the above libraries build a large index that often takes a long time to load the index from memory during nearest neighbor retrieval.

In this paper, we show that the hybrid approximate nearest neighbor indexing and search (HANNIS) approach outperforms existing state-of-the-art ANN method libraries built on the HNSW algorithm. Both indexing and searching for our proposed method have two phases. For indexing, HANNIS first clusters all data points using the kmeans++ algorithm and then arranges the graph into a hierarchical layer of proximity graphs for each cluster along with their cluster center (centroid) information. During the search, HANNIS first loads the index closest to the query and then performs a layer-by-layer search. In Section II we discuss the indexing and searching procedure in detail. HANNIS outperforms the state-of-the-art libraries built on the HNSW algorithm in terms of accuracy for the publicly available SIFT10M dataset [15]. In Section III, we discuss our experimental results in detail. Our HANNIS library is suitable only for vector spaces and does not allow index updating.

A. Related Work

Neighbor-ANN methods for indexing and searching is based on local neighborhood data points. Hierarchical Navigable Small World (HNSW) [1] arranges the graph into a hierarchical layer of proximity graphs, allowing the algorithm to evaluate only a needed portion without being dependent on the entire network. The query process starts at the top layer, where the edges are the longest, and then performs a greedy search within that layer until it reaches a local minimum. The search then switches to the lower layer, where the edges are shorter, and this time the starting point is the previous local minimum and continues until the query is reached. So far, HNSW has provided the best performance in terms of retrieval time among all ANN methods. However, it takes a significant time to build indexes for a large data set. Also, loading the offline index from the database is slow. NSG [3] selects one node and tries to ensure the existence of monotonic pathways connecting it to all other nodes. "Navigating Node" is the term given to this node. The search always begins with the Navigating Node, which makes the search efficient on an NSG. However, NSG is excessively sparse, which hurts search speed. Furthermore, NSG experiences considerable indexing complexity and offers no theoretical guarantee on nearest-neighbor search. Navigating the satellite system graph (NSSG) [4] addresses the problem of theoretical guarantee of NSG by introducing the concept of a satellite system graph (SSG). During the SSG indexing process, an angle **a** is provided for the data set D. The first neighbor of each data point $P \in D$ is the closest data point Q and all other neighbors R of P satisfy the condition $cos\angle RPQ$<$cos\angle a$. NSSG also uses the pruning technique during index building to eliminate the farthest and duplicate data points. The search process starts at the fixed point and proceeds towards the neighbor with the smallest distance from the query. This process repeats until the query is reached. The sparsity of NSSG can be controlled by changing the angle a, making it faster than nongraph-based methods. However, the indexing and memory cost is higher than that of nongraph-based methods.

Neighborhood Graph Tree (NGT) [5] 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 data point to its three nearest neighbors throughout the graph. During the query process, NGT generates a seed using the VP tree [16] 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 bidirectional one [17], and the other is to construct an undirected graph by continuously inserting elements. All neighboring ANN methods suffer from a long index-building time and low retrieval for large deep-descriptor databases [18].

Space Partition ANN methods narrow the entire search space to a smaller group of similar data points. The similarity is defined based on some distance metric, and different partitioning techniques, such as clustering, Voronoi partition, random divided partition, etc., are exploited to separate each similar group. The Faiss library enables efficient partitioning of data in Voronoi cells [15], and the index of each cell is a centroid of that cell. Faiss uses different compression techniques, such as product quantization [19] to compress actual vectors, helping to work faster with large data sets. Data points in each cell can be further indexed using different indexing algorithms such as LSH, HNSW, NSG, etc. In the search algorithm, the centroid with the smallest distance from the query point narrows the search space down to the target cell and performs a similarity search on the target cell. However, for large datasets, a large number of Voronoi regions do not contain data points, and a lot of time is spent searching for empty regions [15].

Data Partition ANN methods for indexing and searching rely on data-driven indexes. One such approach is the kd-tree where the whole data space is partitioned in the dimension with the highest variance recursively until a certain number of data points remain in the leaf. The improvement of the kd-tree proposes the building of multiple randomized kd-trees in parallel by splitting the data into one of the five main dimensions with the highest variance, rather than in the dimension with the highest variance. During the search process, a shared priority queue is used to perform a depth-first search with some heuristic scoring function along all the randomized kd-trees [20]. FLANN combines the randomized kd-tree approach with hierarchical k-means [21] for indexing. In the hierarchical k-means indexing algorithm, k-means clustering is performed recursively until each leaf contains fewer than k-children, and the result is the cluster tree. The search method starts at the root node and crosses the inner node that has the lowest distance from the query to the cluster center. FLANN performance degrades in higher-dimensional feature spaces, as splitting based on a single dimension cannot reflect the entire data set in the kd-tree. In addition, the choice of initial cluster centers in the k-means algorithm makes a huge difference in the final clusters.

Annoy [11] uses a random projection and builds a tree. The space is split into two subspaces at each intermediate node in the tree by a randomly selected hyperplane. This process is done n times to build a forest of trees. One major advantage of Annoy is that it can use static files as indexes and that the indexes can be shared between processes. Additionally, Annoy separates the processes of building an index from loading it, allowing one to share an index as a file and rapidly map it to memory. This helps to reduce the memory footprint. However, Annoy has low performance in terms of speed and accuracy, which they have mentioned in their GitHub library.

Hashing ANN methods for indexing attempt to hash similar input items into the same "buckets" with high probability, and the methods are either data independent, such as locality sensitive hashing (LSH), or data-dependent methods, such

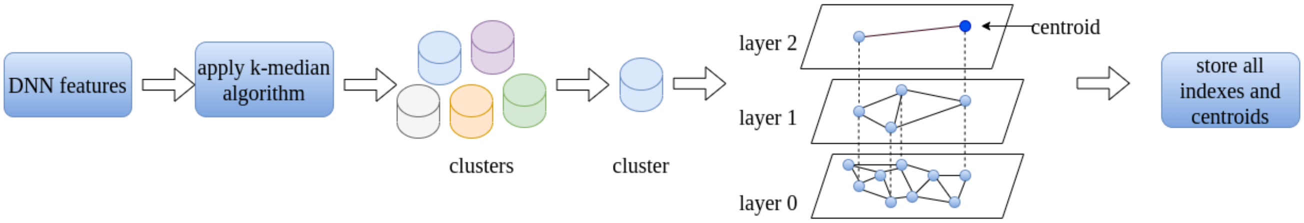

Fig. 2: HANNIS: proposed indexing approach.

as locality-preserving hashing (LPH). LSH-based methods mainly rely on locality-sensitive hash functions with the aim of generating the same hash code for similar input data points and different hash codes for dissimilar data points. The most used hash function for creating hash codes is random linear projections, where the 2-stable distribution (normal distribution) is used to select the random projection parameters. LPH-based methods generate specific hash functions from the entire dataset distribution. The primary purpose of LPH algorithms is to preserve the similarity between the original data space and the hash-coding space. SRS [8] falls under the LSH method, which first generates m 2-stable random projection vectors and calculates the projections for each data point. Then, the m-dimensional projections associated with their respective IDs are assigned a multidimensional index. SRS uses the R-tree [22] index structure, which can incrementally return the $(k + 1)$th nearest neighbor after calculating the kth nearest neighbor. During the search, SRS projects the query with the same m projection vectors to the m-dimensional space. Then an exact k-NN query is performed incrementally on the R-tree index centered on the query projection. The major drawback of SRS is that the points colliding with the query in a part of a hash function in one hash table are ignored since the hash tables are built before searching, even though they are probably close in the vector space. Unlike constructing a hash table with "static" hash functions, QALSH [7] uses a dynamic collision count approach. By building a $B+$-tree on each random projection and performing incremental range queries until the top-k candidates are determined, QALSH pioneered query-aware hash algorithms. It is important to remember that the hash functions are computed intelligently based on the range of possible query executions rather than on a single query point. One drawback of QALSH is that it meets the query accuracy guarantee; additional hash algorithms are needed as the dataset size increases, which requires more time. Furthermore, when new projections are formed, it can be expensive for QALSH to create $B+$ trees on top of each projection to maintain accuracy.

II. Methodology

In this section, we present our proposed method, which is divided into two sections. First, we introduce the algorithm for efficiently preprocessing data and building offline indexes to accelerate the search, as illustrated in Fig. 2. Next, we propose the k -NN retrieval approach, which makes use of the proposed indexing approach.

```
Algorithm 1: BUILD(HANNIS,X,ncls,niter,M,cand)
  Input: multilayer graph HANNIS, data vector X,
         number of clusters ncls, number of iteration to
         find centroids niter, number of established
         connections M, size of dynamic candidate list
         cand
  Output: Update HANNIS inserting all elements
1 clusters,centroids← $KMEANSPP(X, ncls, niter)$
2 foreach element $(cls, cen)$ of $(clusters, centroids)$ do
3     foreach $c$ of $cls$ do
4         $INSERT(HANNIS, c, M, cand)$
5     end
6     $SAVE(HANNIS, cen)$
7 end
```

```
Algorithm 2: SEARCHINDEX(q,centroids,k,n)
  Input: query element $q$, cluster centers centroids,
         number of nearest neighbors $k$, number of
         clusters to load $n$
  Output: $k$ closest neighbors to $q$
1  foreach cen of centroids do
2      find the distance from cen to $q$
3  end
4  nclus ← $n$ closest centroids to $q$
5  foreach iclus of nclus do
6      HANNIS ← $LOADINDEX(iclus)$
7      nclusK← $SEARCH(q, k)$
8  end
9  neighbors ← $k$ nearest neighbors to $q$ in nclusK
10 return neighbors
```

a) Index building: To reduce the search space and accelerate index loading time, we first subdivide the entire data space into n different partitions of $C_1, C_2, C_3, ..., C_n$. Clustering is a technique that can group similar data points, and cluster centroids can be used as representatives of a particular cluster. For this work, we have used the k-means++ algorithm [23], as it chooses the initial cluster center in a way that results in better clustering over k-means. All centroid information and the distances from the centroid to the data points in each cluster are stored and passed to the next phase to build the HNSW index (Algorithm 1 line 2). Once the groups have been calculated, each group is hierarchically arranged. The

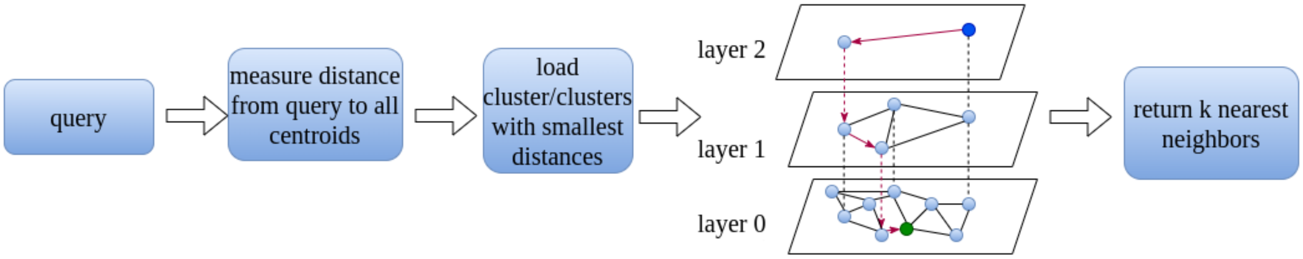
query
measure distance from query to all centroids
load cluster/clusters with smallest distances
layer 2
layer 1
layer 0
return k nearest neighbors

Fig. 3: Proposed HANNIS retrieval pipeline.

graph is arranged in a hierarchical layer of proximity graphs, where the upper layer contains the data points with the longest edges (distance to neighboring data points), and the lower layer contains the data points with the smallest edges (Fig. 2). This layered structure allows the algorithm to evaluate only a needed portion without relying on the entire network and maintains logarithmic complexity. Data points are sequentially added to the graph one at a time. A probability function, $P(L) = F(L, l_m)$, is used to determine the insertion layer of an element. The value L denotes the layer at which an element will be inserted. The probability function normalized by the "level multiplier" l_m, where l_m=0 indicates that vectors are only inserted in layer 0, gives the probability of a vector insertion in a given layer. We achieve the highest performance when we reduce the overlap of shared neighbors between layers. We can reduce overlap by decreasing l_m. However, doing so, as more vectors are moved to the layer 0, increases the average number of search traversals. Therefore, we use a l_m value of $log_2(4M/5)$ that evenly distributes both, where M is the maximum number of established connections at any data point. The starting point for building an index is randomly chosen and a heuristic is used to select neighbors. The fundamental idea of the heuristic is to use a diversity criterion as a filter when choosing the closest neighbors to add to a newly added node in the graph. We skip over a potential neighbor if it is closer to a neighbor that has already been added (which is closer to the new node) than it is to the new node and instead move on to more distant but likely more diverse neighbors. Given that we add links in both directions, the same criterion is also applied to the neighbors of the neighbor nodes. Finally, all indexes are stored with their centroid information so that only a certain index can be loaded based on the query, which in turn saves the index loading time(Algorithm 2 line 5).

b) Improving effectiveness of retrieval: Fig. 3 shows the k -NN retrival approach. During retrieval, the first distances from the query to all centroids are measured and sorted based on their distances. Then the index with the smallest distance from the query to the centroid is loaded for searching. After loading an index, a greedy search is performed in the graph index, and the top k NN is returned. In the Algorithm 2 line 3, the distances between all the centroids and the query are calculated. Then we sort the distances and load the cluster with the smallest query to the centroid distance (Algorithm 2 line 4). A control parameter determines how many indexes to load, and loading multiple indexes improves accuracy, but increases retrieval time (Algorithm 2 line 5). The search within an index starts with the centroid in the upper layer where the edges are the longest, and then a greedy search is used within that layer until it reaches a local minimum (Fig. 3). The search then switches to the lower layer, where the edges are shorter. This time, the starting point is the previous local minimum, and this process continues until the query is reached and the top k-NN to the top k is returned (Algorithm 2 line 8). For multi-index search, each index returns its top k-NN to the query. Then, all the retrieval results are sorted based on their distance to the query. Finally, k -NN are chosen as the final retrieval result(Algorithm 2 line 9). By loading multiple indexes, our multi-index search handles the situation where the query lands on the boundary of a cluster in feature space such that some of its actual nearest neighbors are in a different cluster.

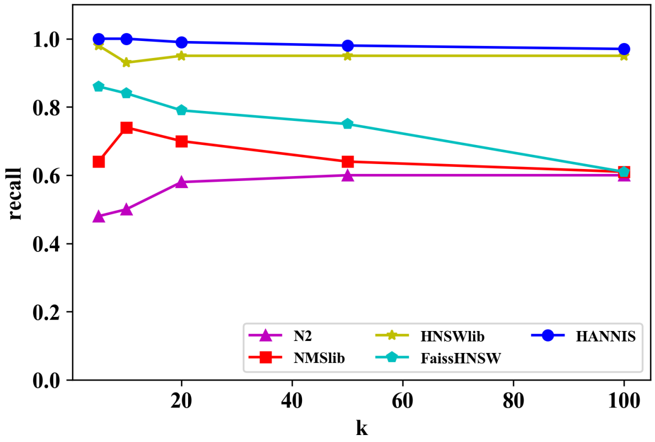

Fig. 4: Recall@k for DOTA 2.0 dataset with 2,697,873 instances of 1024 dimensional floats for five methods N2, Nmslib, Faiss HNSW, HNSWlib, and HANNIS.

III. Experiments

For all of our experiments, we have used the SIFT10M benchmark data set with 10 million instances and 128 dimensions [24], and the DOTA 2.0 data set with 1,024 dimensions and 2,7 million instances [25]. For the subsequent data set, we extracted deep features using Faster-RCNN [26] with Resnet50 [27] as the backbone and the Detectron2 [28] code baseline.

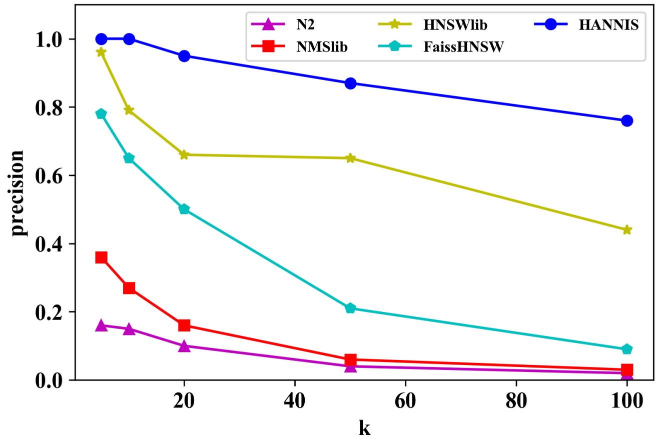

Fig. 5: Precision@k for DOTA 2.0 dataset with 2,697,873 instances of 1024 dimensional floats for five methods N2, Nmslib, Faiss HNSW, HNSWlib, HANNIS.

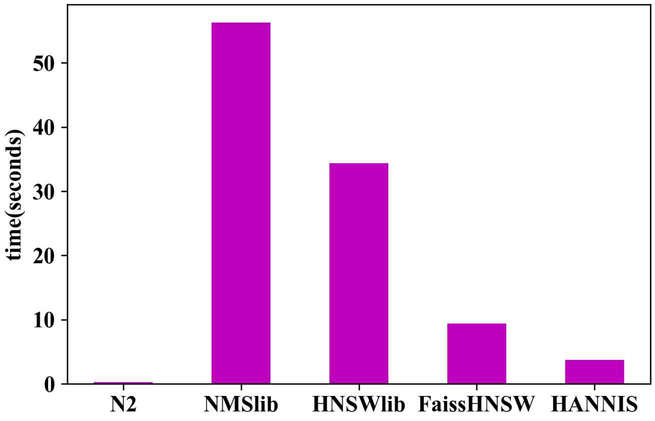

Fig. 7: Index loading times into memory for DOTA 2.0 dataset with 2,697,873 instances of 1024 dimensional floats for $k = 100$ for five methods N2, Nmslib, Faiss HNSW, HNSWlib, HANNIS.

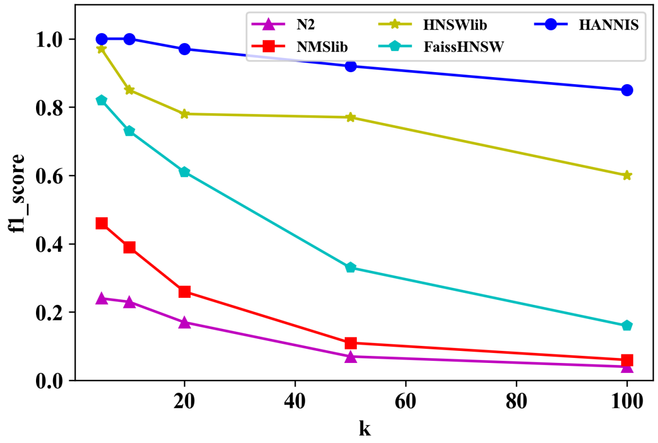

Fig. 6: F1-score@k for DOTA 2.0 dataset with 2,697,873 instances of 1024 dimensional floats for five methods N2, Nmslib, Faiss HNSW, HNSWlib, HANNIS.

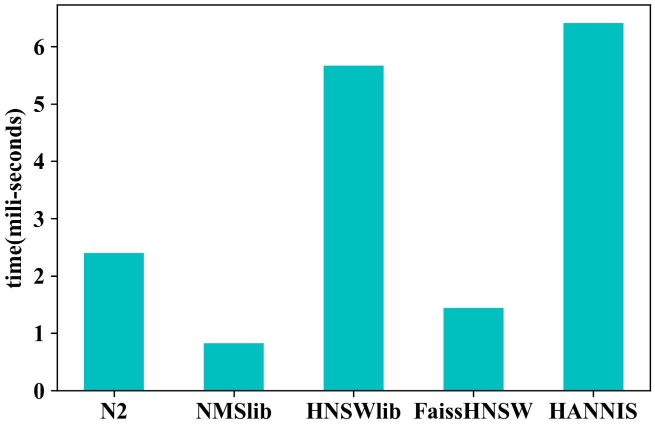

Fig. 8: Retrieval times for the DOTA 2.0 dataset with 2,697,873 instances of 1024 dimensional floats for $k = 100$ for five methods N2, Nmslib, Faiss HNSW, HNSWlib, HANNIS.

Performance analysis We measured the performance of our HANNIS method with four different state-of-the-art methods built on the HNSW algorithm. Five performance measurement metrics have been used to evaluate the performance of each method: recall@k, precision@k, f1-score@k, index loading time, and retrieval time. Fairness to all, we used the same parameters for all search methods, including HANNIS. The number of established connections and the size of the dynamic candidate list was set at 16 and 200, respectively, throughout all experiments. 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.

Recall @ k is measured as the percentage of closest neighbors in the top k closest in k, k in the $[5, 10, 20, 50, 100]$ retrieval set of the 5 methods compared to the truth of the ground in k, GT_k, a brute-force search for nearest neighbors in Euclidean space: $R_k = |M_k \cap GT_k|/|GT_k|$. M_k is the set of descriptors recovered by the method M, $M \in [$ N2, NMSlib, HNSWlib, FaissHNSW, HANNIS], and $|M_k \cap GT_k|$ is the number of true positives, the number of nearest neighbors recovered by the method M_k that match GT_k. **Precision @ k** is measured as the fraction of the retrieval set that is relevant to the query: $P_k = |M_k \cap GT_k|/|M_k|$. **F1-score** is measured by $F_k = 2 * R_k * P_k/(R_k + P_k)$. **The load time of the index** measures the time it takes to load the index from memory for retrieval. **Retrieval time** defines the time to retrieve the nearest neighbor k for a single query.

DOTA2.0 dataset performance measures for five methods are compared in Fig. 4. The set contains 2,697,873 instances of 1024-dimensional object deep feature descriptors extracted from DOTA2.0 using ResNet, and we are evaluating the method most suitable for unknown class discovery in the DOTA2.0 dataset. Our criteria are that the recall@k needs to remain

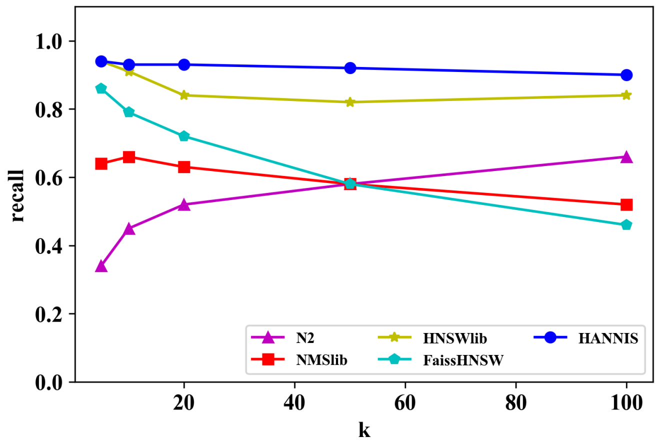

Fig. 9: Recall @ k for the SIFT dataset with 10 million instances of 128-dimensional integers for five methods N2, NMSlib, FaissHNSW, HNSWlib, HANNIS.

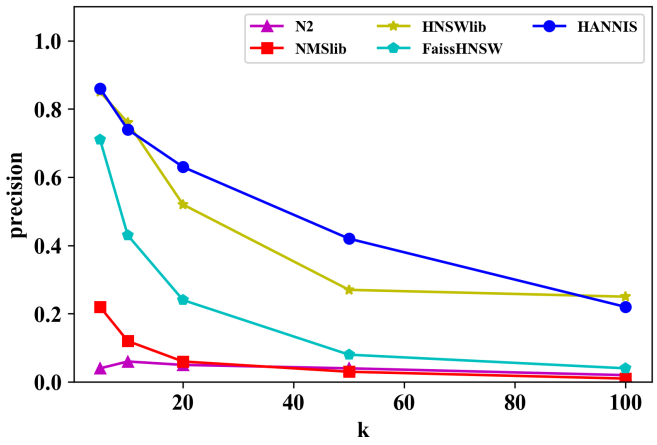

Fig. 10: Precision@k for SIFT dataset with 10 million instances of 128-dimensional integers for five methods N2, NMSlib, FaissHNSW, HNSWlib, HANNIS.

consistent as k increases while keeping the index loading time and retrieval time comparable to the state-of-the-art. We tried different numbers of clusters during the k-means++ clustering phase and got the best results in 20 clusters. Fig. 4 recall@k-retrievals demonstrate HANNIS consistent retrieval recall@k slightly better than HNSWLib, while the HANNIS index loading time is much faster in Fig. 7, and retrieval times are comparable in Fig. 8. The performance of NMSlib degrades rapidly for $k > 5$ (Fig. 4), and while its retrieval times are the shortest (Fig. 8), its index loading times are the highest Fig. 7, and the method is not suitable. N2 has comparably fast index loading times (Fig. 7) and retrieval times (Fig. 8), but performance quickly degrades for the larger retrieval set (Fig. 4). FaissHNSW has a comparably higher index loading time (Fig. 7) and faster retrieval time (Fig. 8 but performance quickly degrades for k>10. Precision@k-retrievals in Fig. 5 and F1-score@k-retrieval in Fig. 6 shows HANNIS out-performs all the methods at $allk \in [5, 10, 20, 50, 100]$. HNSWlib and FaissHNSW do well in $k = 5$, but their performance degrades quickly for higher retrievals in Fig. 5 and Fig. 6. N2 and NMSlib have consistently low precision and low F1 score for all k in Fig. 5 and Fig. 6. HANNIS shows to be the most suitable algorithm for unknown class discovery for the DOTA 2.0 dataset.

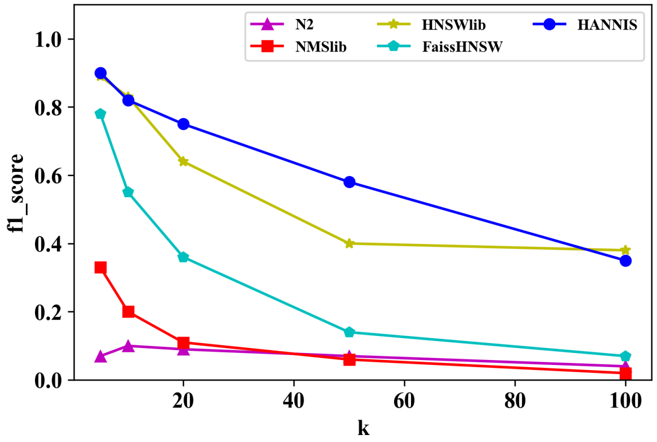

Fig. 11: F1 score @k for the SIFT dataset with 10 million instances of 128-dimensional integers for five methods N2, NMSlib, FaissHNSW, HNSWlib, HANNIS.

SIFT dataset performance measures for five methods are compared in Fig. 9. The set contains 10,000,000 instances of 128-dimensional integer SIFT [24], [29] image descriptors extracted from Caltech-256 41 $\prod$ 41 whole image patches. As the features were extracted from image patches and not object regions, our criteria here are to evaluate the most suitable method to match small image regions for large values of k for the “needle in haystack” search scenarios while keeping the index loading time and retrieval time in check for $k = 100$ [30] (Fig. 9). We tried different numbers of clusters during the k-means++ clustering phase and got the best results in 15 clusters. Recall@k-retrievals demonstrates interesting behavior for all methods that differ from DOTA2.0. HANNIS is now consistently dominating in the effectiveness of retrieval at **all** $k \in [5, 10, 20.50.100]$, and consistently better than HNSWlib for $k > 5$ with smaller index loading times (Fig. 12) at the price of much higher retrieval time (Fig. 13). FaissHNSW has higher index loading times Fig. 12 and comparable retrieval times Fig. 8, but the performance of FaissHNSW quickly degrades for $k > 5$ Fig. 4 and we do not consider the method appropriate for this scenario. We also observe an interesting behavior of N2 in Fig. 9: the effectiveness of the indexing method is *improving* with larger k and approaching the effectiveness of $HNSWlib$ for $k = 100$ at low index loading time (Fig. 12) and low retrieval time (Fig. 13). The performance of NMSlib improves for $k = 10$ and then decreases, even if the retrieval times are the fastest for $k = 100$ Fig. 13. The precision@k retrieval in Fig. 10 shows the dominant performance of HANNIS over N2, NMSlib, and FaissHNSW at **all** k. HANNIS performs better

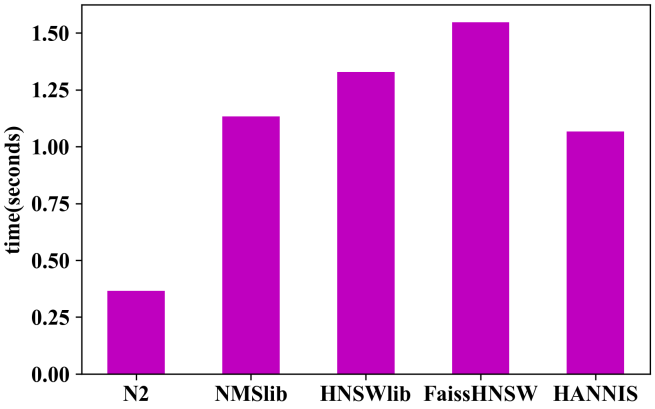

Fig. 12: Index loading times in memory for the SIFT dataset with 100,000 128-dimensional integer instances for $k = 100$ for five methods N2, NMSlib, FaissHNSW, HNSWlib, and HANNIS.

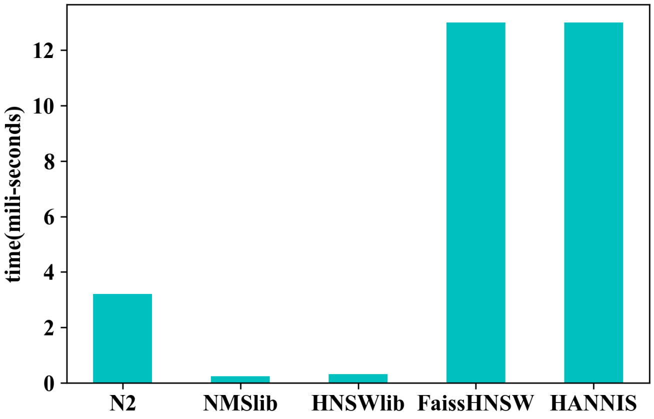

Fig. 13: Retrieval time for the SIFT dataset with 10,000 128-dimensional integer instances for $k = 100$ for five methods N2, NMSlib, FaissHNSW, HNSWlib, and HANNIS.

than HNSWlib for $k > 10$, but interestingly HNSWlib catches up with HANNIS for $k = 100$ in Fig. 10. F1-score@k retrieval shows a similar trend as Precision@k for the SIFT10M dataset in Fig. 11. HANNIS outperforms N2, NMSlib and FaissHNSW at **all** $k \in [5, 10, 20.50.100]$ in Fig. 11. The performance of HNSWlib degrades for $k > 10$ but eventually catches up to HANNIS at $k = 100$ in Fig. 11. In general, HANNIS proved to be the most robust algorithm for the "needle in the haystack" similar patch search in the SIFT Caltech-256 data set.

Next, we analyze the five methods index load timing and retrieval timing on a log scale for 2.7 million DOTA2.0 1024-dimensional float vectors (2764 million floats), 10 million SIFT10M 128-dimensional integer vectors (1,280 million integers) *per method* to uncover any common trends. DOTA2.0 is approximately nine times the size of SIFT10M. Fig. 14 compares the index loading time, and NMSlib, HNSWlib, FaissHNSW, and HANNIS index load time is proportional to the size of the dataset, and HANNIS has the best overall

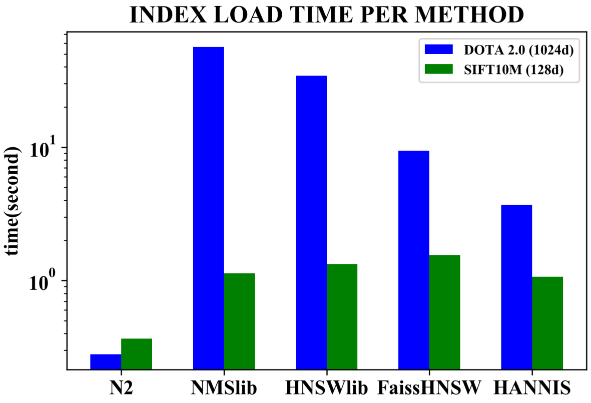

Fig. 14: Index loading time per method for 5 approaches on 2 datasets.

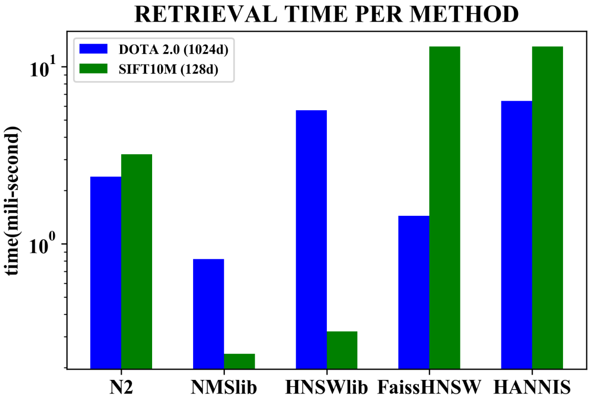

Fig. 15: Retrieval time for 5 approaches for 2 datasets.

index load timing. N2 has the lowest index loading time, and it does not correlate to data set size in instances and in feature dimension, and type. Fig. 15 compares retrieval time per method for $k = 100$. For N2, FaissHNSW, and HANNIS methods, retrieval time corresponds to the number of instances in the dataset, and HANNIS seems to do better on high-dimensional descriptors than the comparable methods. For HNSWlib, the retrieval time is directly proportional to the dataset size, and it is hard to interpret the rule for NMSlib methods.

IV. Conclusion

The approximate nearest neighbor search approach mitigates the high cost of brute force k-Nearest Neighbor Search in large and high-dimensional data. In this paper, we have proposed a nearest-neighbor indexing and search method (HANNIS) to index and retrieve similar content from high-dimensional deep-descriptor data set, and demonstrated method's effectiveness for two real scenarios: unknown class discovery and whole-image feature search. HANNIS efficiently clusters all the data points using kmeans++ and then builds a Hierarchical Navigable Small World (HNSW) graph index for each cluster. During retrieval, HANNIS only loads the indexes that are most similar to the query. Recall exceeds that of all existing state-of-the-art libraries built on the HNSW algorithm up to 100. HANNIS is up to 18 times faster than state-of-the-art libraries in terms of index loading time. Retrieval times are similar to those of state-of-the-art libraries for searching in vector space.

Acknowledgement

This work is partially supported by the NAVAIR SBIR N68335-18-C-0199. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government paper references.

References

[1] Yu A Malkov and Dmitry A Yashunin. 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.

[2] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586, 2011.

[3] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

[4] Cong Fu, Changxu Wang, and Deng Cai. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[5] Masajiro Iwasaki. Ngt: Neighborhood graph and tree for indexing, 2015.

[6] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. *Advances in neural information processing systems*, 28, 2015.

[7] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 9(1):1–12, 2015.

[8] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proceedings of the VLDB Endowment*, 2014.

[9] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[10] Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence*, 36(11):2227–2240, 2014.

[11] Erik Bernhardsson. *Annoy: Approximate Nearest Neighbors in C++/Python*, 2018. Python package version 1.17.1.

[12] GeonHee Lee. *TOROS N2 - lightweight approximate Nearest Neighbor library which runs fast even with large datasets*, 2017. Python package version 0.1.7.

[13] Leonid Boytsov and Bilegsaikhan Naidan. Engineering efficient and effective non-metric space library. In *International Conference on Similarity Search and Applications*, pages 280–293. Springer, 2013.

[14] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.

[15] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. Revisiting the inverted indices for billion-scale approximate nearest neighbors. *CoRR*, abs/1802.02422, 2018.

[16] Masajiro Iwasaki and Daisuke Miyazaki. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355*, 2018.

[17] Masajiro Iwasaki. Pruned bi-directed k-nearest neighbor graph for proximity search. In *International Conference on Similarity Search and Applications*, pages 20–33. Springer, 2016.

[18] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2019.

[19] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[20] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.

[21] Keinosuke Fukunaga and Patrenahalli M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE transactions on computers*, 100(7):750–753, 1975.

[22] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[23] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.

[24] Xiping Fu, Brendan McCane, Steven Mills, Michael Albert, and Lech Szymanski. Auto-jacobin: Auto-encoder jacobian binary hashing. *CoRR*, abs/1602.08127, 2016.

[25] Gui-Song Xia, Xiang Bai, Jian Ding, Zhen Zhu, Serge Belongie, Jiebo Luo, Mihai Datcu, Marcello Pelillo, and Liangpei Zhang. Dota: A large-scale dataset for object detection in aerial images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3974–3983, 2018.

[26] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[28] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. https://github.com/facebookresearch/detectron2, 2019.

[29] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[30] Jelena Tesic and B. S. Manjunath. Nearest neighbor search for relevance feedback. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2003), 16-22 June 2003, Madison, WI, USA*, pages 643–648. IEEE Computer Society, 2003.