Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs

Diykanbaev Talant (57545625)

Mahmud Jawad (57274243)

Mukhambetzhanuly Imangali (57330097)

Halimatmadja Samantha Abbygail (57259633)

Wijaya Jessica (57201804)

City University of Hong Kong

2024/25 Semester A

SDSC 3001 Big Data : The Art & Science of Scaling

Table of Contents

1. Introduction

In today's digital world, with the rapid growth of data, we need efficient and scalable methods to retrieve relevant information. K-Nearest Neighbor Search (K-NNS) is frequently used in domains such as machine learning, image recognition, and document retrieval to find the most similar results to a query. However, K-NNS has scaling issues because it becomes expensive and time-consuming to calculate distances to every point in huge datasets, especially in high-dimensional spaces. Therefore, a more flexible technique, exchanging exact matches for maximum efficiency, Approximate Nearest Neighbour Search (K-ANNS), has been applied to handle these difficulties. Despite this, K-ANNS still suffers performance problems in many datasets, including low-dimensional or clustered data.

With our main problem discussed, in this study, we present the Hierarchical Navigable Small World (Hierarchical NSW) model, which incorporates a layered structure to improve routing efficiency and scalability. This method allows searches to begin in upper layers with larger connections and progress to more refined lower layers as needed, increasing both speed and accuracy. Hierarchical NSW tries to provide logarithmic scalability across various dataset architectures by employing probabilistic layer assignment and limiting node connections. Finally, we believe that this approach can be used to improve search performance and flexibility across a variety of data settings when compared to traditional search systems.

2. Main Problem

Starting at an entrance point and moving throughout the graph, selecting nearby nodes with the smallest distances until a stopping point is reached, most algorithms for searching in K-Nearest Neighbor graphs usually use a greedy routing strategy. A rapid method to find nearest neighbors and a close approximation to Delaunay graphs are K-NN graphs. Still, drawbacks include steps scaling with dataset size using a power law and a possible lack of global connectivity, most especially in clustered data. Combining approaches, such as using kd-trees or product quantization in the first stage of candidate selection, try to tackle these challenges. Known as Navigable Small World (NSW) methods, algorithms such as Metricized Small World (MSW) build navigable graphs with logarithmic or polylogarithmic hops scaling, hence enhancing search efficiency. NSW creates graphs ensuring network connectivity by connecting elements to their M nearest neighbors in both directions, hence enabling logarithmic hop scaling. NSW highlights performance and parallelizability on some datasets and distributed systems, but its polylogarithmic complexity scaling may cause performance problems with low-dimensional data, therefore undermining its advantages over tree-based techniques. Influenced by spatial models, including Kleinberg's and scale-free models, each with their complexities and demands, the concept of navigable small world networks, with an eye toward successful graph routing, is shaped. Though juggling polylogarithmic search complexity, this has resulted in NSW as a distributed, flexible model fit for many data settings.

Navigable Small World (NSW) search complexity is optimized by a thorough analysis of the routing process marked by the "zoom-out" and "zoom-in" phases. Starting with a low-degree node, the greedy method progressively raises node degrees until connection lengths correspond to the query distance, hence possibly causing entrapment in far-off false minima. Starting the search from high-degree nodes, like early NSW structure nodes, moves directly into the "zoom-in" phase, improving successful routing probability and performance on low-dimensional data. Despite these improvements, the polylogarithmic scalability of a single greedy search remains constant and performs less well on high-dimensional data than Hierarchical NSW. The polylogarithmic complexity results from the product of average greedy hops and node degrees, both scaling logarithmically, as the search often encounters hubs during growth and hub connections increase logarithmically with network size, so generating an overall polylogarithmic complexity.

3. Proposed Solution

Hierarchical Navigable Small World (HNSW) is a graph-like data structure that merges features of the Navigable Small World (NSW) algorithm with the idea of a probability skip list. Within the HNSW framework, the system maintains a hierarchy of levels allowing efficient searching for nearest neighbours in high dimension environments.

3.1. Navigable Small World (NSW)

Navigable Small World (NSW) emphasises the creation of graphs with logarithmic or polylogarithmic hops during greedy traversal. The NSW graph enhances routing efficiency by connecting elements and their M nearest neighbours in both directions, hence bridging centres of networks. NSW aims to enable logarithmic scaling of hop counts in greedy routing and maintain general graph connection. Navigable Small Worlds run on the theory that any other node can be reached in a certain distance of "hops". Once an entrance point is decided upon—which might be selected at random, depending on heuristics, or by another method, while the surrounding nodes are investigated to find those that are closer than the current node. The search keeps on the closest adjacent nodes and repeats until no more closer nodes show themselves. The M value dictates the number of connections a new node will create with its closest neighbours (as illustrated above with M=2), hence directing graph construction.

Even though it requires more memory and has slower insertion speeds, a higher M results in a more linked and dense graph. As nodes are added to the network, it searches for the nearest M nodes and creates links in both directions with them. Like social networks, where some persons are connected to many others while others are only connected to a handful, network nodes will have varied degrees of connectivity. An extra option known as M, max determines a node's maximum number of edges, as too many edges can have an impact on performance (Miesle, 2023).

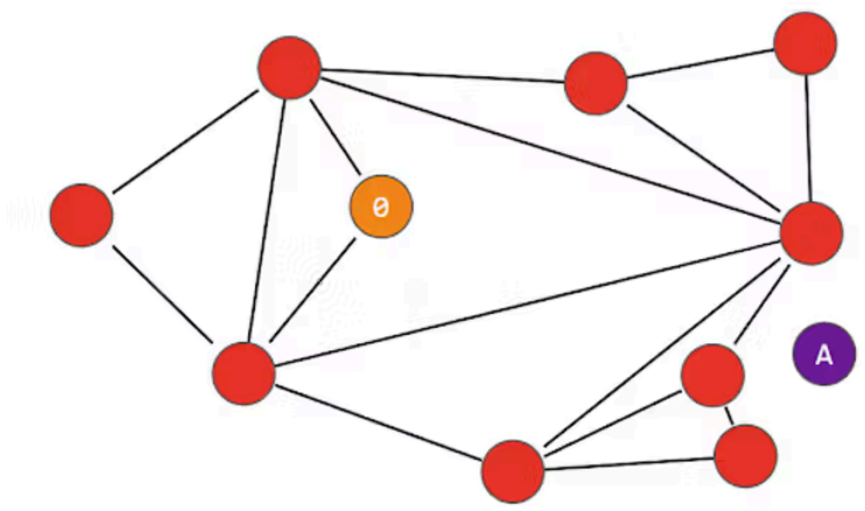

Figure 1. NSW Graph Search Methods

3.2. Probability Skip List

A skip list is a type of data structure that enables quick searching in a sorted series of elements by bypassing several elements in each iteration. The skip list's efficiency is boosted by introducing randomness through the probability skip list, which determines the number of elements to skip at each level. The arrangement of a skip list enables efficient insertion and querying operations with an average time complexity of O(log n), where the operation time grows in a logarithmic manner based on the data size, instead of linearly, poly logarithmically, or worse. Every node at the bottom of a skip list is linked to the next node in a sequential order. Adding more layers causes nodes in the sequence to be "skipped". Skip lists are often built using probability, the likelihood of a node appearing in a layer reduces as the layer progresses, resulting in fewer nodes in higher layers. When searching for a node, the algorithm begins at the "head" of the top layer and progresses to the next element that is equal to or greater than the target. At this stage, the search item is either discovered or moves down to the next layer, repeating until you find the search node or its neighbouring nodes. In skip lists and similarity search, "greater than" is equivalent to "more similar". Inserting requires finding the layer at which the node should be placed, moving to the first node that is considered "larger than" while also keeping note of the last node before the "larger than" node. After locating the "greater than" node, it changes the previous node's pointer to point to the new node, and the new node points to the "greater than" node. It continues to the subsequent layer below until the node is present in every layer. Removing items follows a comparable reasoning to finding and modifying pointers (Miesles, 2023).

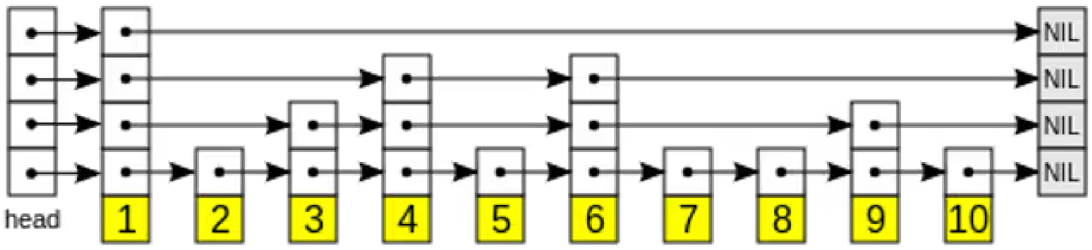

Figure 2. Skip List Search Methods

3.3. Hierarchical Navigable Small World (HNSW)

The Hierarchical NSW algorithm employs a stratified method to organize links by their size, facilitating effective exploration in a multi-tiered graph. Logarithmic scalability is attained by assessing a set fraction of links for every item, irrespective of the network's size. The exploration starts at a higher level with longer connections during a "zoom-in" stage, traveling through components until it reaches a point of minimal proximity before shifting to lower levels with shorter connections and repeating the cycle. Ensuring a stable amount of connections per element across all layers results in consistent scaling complexity, which improves route optimization in a small global network that is simple to explore.

Elements are assigned a level that determines the highest layer they can be placed on in order to form the layered structure, and proximity graphs are then generated for elements within each layer. Using a probability distribution that declines exponentially with level produces a logarithmic increase in predicted layers. The search strategy employs an incremental greedy method, beginning with the top layer and progressing to the lowest layer. Unlike NSW, Hierarchical NSW does not require randomization of items before adding them; instead, unpredictability is introduced through level randomization, allowing for gradual indexing even with shifting data distributions.

The Hierarchical NSW idea is structured similarly to a 1D probabilistic skip list, but instead of linked lists, proximity graphs are used, making distributed approximation search structures easier to design. When inserting elements, a heuristic considers potential distances to establish a variety of connections, promoting a connected component even in highly clustered data. This rule selects links based on their proximity to the primary element, ensuring efficient neighbor selection and regulating connection amounts to improve search performance.

The basic form of Hierarchical NSW proximity graphs, also known as sparse neighborhood graphs,' has been used for proximity graph search. Comparable methodologies have been examined in FANNG algorithms, which focus on exact routing characteristics over sparse neighborhood graphs. These methods improve search efficiency by carefully selecting links and emphasizing globally connected components, which are critical for obtaining peak performance in various data structures.

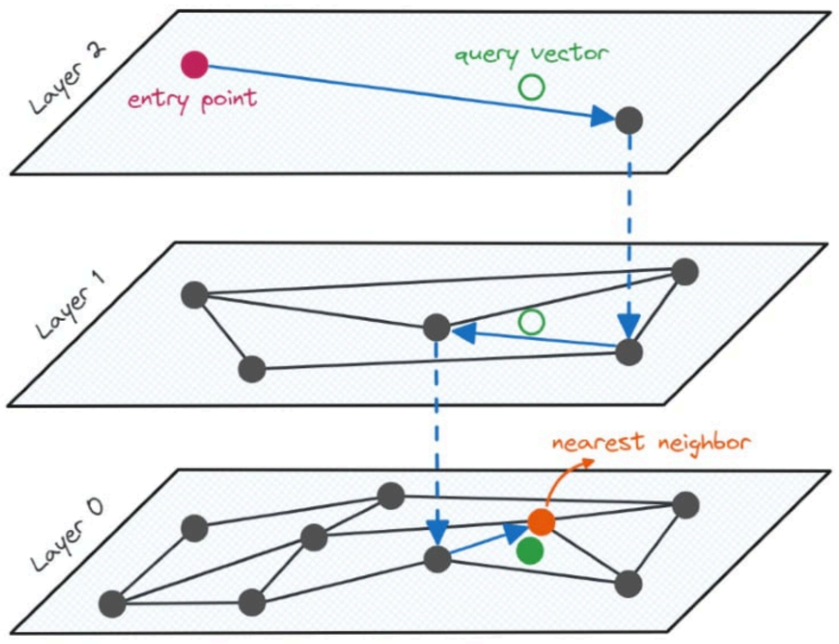

Figure 3. HNSW Graph Search Methods

3.3.1. Algorithm Descriptions for HNSW

The proposed solution includes 5 algorithms to create the HNSW graph, the algorithms are described below:

3.3.1.1. Insert Node

Outlines how to add a new element q to the HNSW (Hierarchical Navigable Small World) multilayer graph. It initially finds the entry point ep in the graph and creates a list W to save nearby components. Starting from the highest layer L in the entry point, the algorithm generates a new level l by means of random dissemination. Searching inside each layer to identify possible elements, one starts from the top layer and descends to the level of the new element. Connections are created in both ways between neighbors and q at the same layer following the selection of adjacent elements using a certain method. Modifications are applied to the connections if the quantity of connections surpasses the predefined maximum limit Mmax. The process continues over every level, modifying the starting point and maybe allocating it to the newest element q should its level surpass the level of the original starting point. This method effectively combines the new element into the hierarchical navigable small world structure, therefore strengthening links and maintaining graph integrity during the insertion process.

3.3.1.2. Search-Layer

Explains how to find the ef nearest neighbours to query element q in a particular layer lc of the HNSW multilayer graph. First, the algorithm initializes groups for visited items v, potential candidates C and the evolving list of closest neighbors W, all originating from the entry point ep. It then iterates over the set of candidates C until all elements have been evaluated, or the desired number of nearest neighbors have been found. The algorithm updates sets C and W for each candidate element c by examining neighboring elements at layer lc. The element will also be inserted into C and W if it is closer to q than the most distant element in W or when the current number of neighbors in W is below ef. If the amount in W exceeds ef, this algorithm eliminates the most-distant element from W against q. The algorithm terminates with a return of the set ef nearest neighbors W from input element q in the targeted layer lc.

3.3.1.3. Select-Neighbors-Simple

This algorithm focuses on selecting the M nearest elements to a base element q from a set of candidate element C. Based on their proximity, the algorithm will generate the M nearest elements from the candidate set C to the base element q. By implementing this selection process, we are aiming to efficiently identify and return the specified number of nearest neighbors to the base element q from the provided candidate elements C.

3.3.1.4.Select-Neighbors-heuristic

Chooses M elements using a heuristic method from a candidate set C within a certain layer lc, with options to expand candidates and maintain pruned connections. The algorithm starts by creating two sets: R, which holds the chosen elements, and W, which contains the potential candidates to be selected. When the extendCandidates flag is turned on, the algorithm enlarges the candidate list by taking into account neighbors of elements in C. It continues to go through the working queue W until the desired number of elements is chosen. After that, the algorithm will continue by comparing the distance between each element e and the base element q to elements in R. e is included in R if e is closer to q than any element in R; otherwise, it finds place in the Wd queue for candidate elimination. If the keepPruned Connections flag be set, the algorithm might include previously rejected Wd to the chosen item connections in R. This procedure keeps on until the heuristic criterion is used to choose M items; at that time the algorithm generates the set of M elements chosen by the heuristic.

3.3.1.5 KNN-Search

Conducts a search for k-nearest neighbors in the HNSW multilayer graph structure. The algorithm requires the multilayer graph HNSW, a query item q, the number K of nearest neighbors to provide, and the ef size of the dynamic candidate list as input. A set W locates the entry point ep in the graph and maintains track of the closest members. Starting at the top layer L of the entrance point, the algorithm descends the levels searching each one to find possible items. After each search, the entry point is changed to the closest element found. Finally, the algorithm searches layer 0 to further examine the data and presents the K items in set W that are the closest to the query element q. This step ensures that the algorithm correctly locates and delivers the specified number of nearest neighbors to query element q in the HNSW multilayer graph structure.

3.3.2. Experiment Code and Discussion

Algorithm for Proposed Solution Using Python:

```python
import numpy as np
from heapq import heappush, heappop, heapify, nlargest, nsmallest
from copy import deepcopy
from math import log
from time import time

class HNSW:
    def __init__(self, dim, max_elements, ef_construction=200, M=32):
        self.dim = dim
        self.max_elements = max_elements
        self.ef_construction = ef_construction
        self.M = M
        self.M_max = M
        self.M_max0 = 2*M
        self.m_L = 1/log(M)
        self.data = []
        self.layer = []
        self.links = []
        self.enter_point = None

    def _get_random_layer(self):
        layer = int(-log(np.random.rand()) * self.m_L) + 1
        return layer

    def _distance(self, vec1, vec2):
        return np.linalg.norm(vec1-vec2)

    def _insert(self, qvec):
        if len(qvec) != self.dim or len(self.data) >= self.max_elements:
            return
        self.data.append(qvec)
        self.layer.append(self._get_random_layer())
        self.links.append([])
        q = len(self.data) - 1
        if self.enter_point == None:
            self.enter_point = q

        W = [] # list for the currently found nearest elements (W is a max
heap)
        ep = [self.enter_point] # get enter point for hnsw
        L = self.layer[self.enter_point] # top layer for hnsw
        l = self.layer[q] # new element's level
        for l_c in range(L, l, -1): # l_c ← L ... l+1
            W = self._search_layer(qvec, ep, ef=1, l_c=l_c)
            ep = [point for dist, point in nlargest(1, W)] # get the
nearest element from W to q
        for l_c in range(min(L, l), -1, -1): # l_c ← min(L, l) ... 0
            W = self._search_layer(qvec, ep, self.ef_construction, l_c)
            neighbors = self._select_neighbors_simple(q, W, self.M) # alg.
3
            #neighbors = self._select_neighbors_heuristic(q, W, self.M,
l_c) # or alg. 4
            self._add_bidirectional_connections(neighbors, q, l_c)

            for e in neighbors: # shrink connections if needed
                eConn = self._get_neighborhood_at_layer(e, l_c)
                eConn = [(-self._distance(qvec, self.data[point]), point)
for point in eConn]
                heapify(eConn) # eConn is a max heap
                if len(eConn) > (self.M_max if l_c != 0 else self.M_max0):
# shrink connections of e

# if l_c = 0 then M_max = M_max0
                    eNewConn = self._select_neighbors_simple(q, eConn,
self.M) # alg. 3
                    #eNewConn = self._select_neighbors_heuristic(q, eConn,
self.M_max, l_c) # or alg. 4
                    self._update_neighborhood_at_layer(e, l_c, eNewConn)

            ep = [point for dist, point in W]

        if l > L:
            self.enter_point = deepcopy(q)

    def _search_layer(self, qvec, ep, ef, l_c):
        #v = deepcopy(ep) # set of visited elements
        v = {}
        for x in ep:
            v[x] = True
        C = [(self._distance(qvec, self.data[point]), point) for point in
ep] # set of candidates
        W = [(-self._distance(qvec, self.data[point]), point) for point in
ep] # dynamic list of found nearest neighbors
        heapify(C) # C is a min heap
        heapify(W) # W is a max heap

        while C:
            dist_c_to_qvec, c = heappop(C)
            dist_f_to_qvec, f = -W[0][0], W[0][1]
            if dist_c_to_qvec > dist_f_to_qvec:
                break # all elements in W are evaluated
            for e in self._get_neighborhood_at_layer(c, l_c): # update C
and W
                if e not in v:
                    #v.append(e)
                    v[e] = True
                    dist_f_to_qvec, f = -W[0][0], W[0][1]
                    if self._distance(self.data[e], qvec) < dist_f_to_qvec
or len(W) < ef:
```

```
                        heappush(C, (self._distance(qvec, self.data[e]),
e))
                        heappush(W, (-self._distance(qvec, self.data[e]),
e))
                        if len(W) > ef:
                            heappop(W)
        return W

    def _select_neighbors_simple(self, q, C, M):
        return [point for dist, point in nlargest(M, C)] # C is a max heap

    def _select_neighbors_heuristic(self, q, C, M, l_c, extendCandidates=0,
keepPrunedConnections=0):
        R = [] # R is a min heap
        W = [(-dist, point) for dist, point in C] # Turning from max heap
to min heap
        heapify(W) # working queue for the candidates
        if extendCandidates: # extend candidates by their neighbors
            for e in C:
                for e_adj in self._get_neighborhood_at_layer(e[1], l_c):
                    if e_adj not in W:
                        heappush(W, (self._distance(self.data[q],
self.data[e_adj]), e_adj))
        W_d = []
        while len(W) > 0 and len(R) < M:
            dist_q_to_e, e = heappop(W)
            if len(R) == 0:
                heappush(R, (self._distance(self.data[q], self.data[e]),
e))
                continue
            dist_q_to_r, r = heappop(R)
            if dist_q_to_e < dist_q_to_r:
                heappush(R, (self._distance(self.data[q], self.data[e]),
e))
            else:
                heappush(W_d, (self._distance(self.data[q], self.data[e]),
e))
        if keepPrunedConnections: # add some of the discarded connections
from W_d
            while len(W_d) > 0 and len(R) < M:
                dist, w_d = heappop(W_d)
                heappush(R, (dist, w_d))
        return [point for dist, point in R]

    def _get_neighborhood_at_layer(self, q, layer):
        neighbors_at_layer = [link[0] for link in self.links[q] if link[1]
== layer]
        return neighbors_at_layer

    def _update_neighborhood_at_layer(self, q, layer, W):
        self.links[q] = [link for link in self.links[q] if link[1] !=
layer]
        for w in W:
            self.links[q].append((w, layer))

    def _add_bidirectional_connections(self, neighbors, q, layer):
        for neighbor in neighbors:
            if neighbor == q:
                continue
            if (neighbor, layer) not in self.links[q]:
                self.links[q].append((neighbor, layer))
            if (q, layer) not in self.links[neighbor]:
                self.links[neighbor].append((q, layer))

    def fit(self, X):
        for vec in X:
            self._insert(vec)

    def _k_nn_search(self, query_vec, K, ef):
        W = [] # set for the current nearest elements (W is a max heap)
        ep = [self.enter_point] # get enter point for hnsw
        L = self.layer[self.enter_point] # top layer for hnsw
        for l_c in range(L, 0, -1):
            W = self._search_layer(query_vec, ep, ef=1,  l_c=l_c)
            ep = [point for dist, point in nlargest(1, W)] # get nearest
element from W to q
        W = self._search_layer(query_vec, ep, ef=ef, l_c=0)

        return [(point, float(-dist)) for dist, point in nlargest(K, W)]

def ivecs_read(fname):
    a = np.fromfile(fname, dtype='int32')
    d = a[0]
    return a.reshape(-1, d + 1)[:, 1:].copy()
```

```
def fvecs_read(fname):
    return ivecs_read(fname).view('float32')

if __name__ == "__main__":
    hnsw = HNSW(dim=128, max_elements=10000)

    base = fvecs_read("siftsmall/siftsmall_base.fvecs")
    groundtruth = ivecs_read("siftsmall/siftsmall_groundtruth.ivecs")
    learn = fvecs_read("siftsmall/siftsmall_learn.fvecs")
    query = fvecs_read("siftsmall/siftsmall_query.fvecs")
    print("Base shape:", base.shape)
    print("Ground truth shape:", groundtruth.shape)
    print("Learn shape:", learn.shape)
    print("Query shape:", query.shape)

    #data = np.random.rand(10, 2)
    #hnsw.fit(data)

    start = time()
    hnsw.fit(base)
    end = time()
    print("Fitting data:", end - start, "seconds.")

    #query_vec = np.random.rand(2)
    #neighbors = hnsw._k_nn_search(query_vec, K=5, ef=hnsw.ef_construction)
    for i in range(len(query)):
        start = time()
        neighbors = hnsw._k_nn_search(query[i], K=10,
ef=hnsw.ef_construction)
        end = time()
        #print("Querying:", end - start, "seconds.")
        #print("Nearest neighbors:", [point for point, dist in neighbors])
        #print("Ground truth:", groundtruth[i][:10])
        correct_ans = 0
        for point, dist in neighbors:
            if point in groundtruth[i][:10]:
                correct_ans += 1
        print(correct_ans)
```

The above python code is an implementation of HNSW (Hierarchical Navigable Small World) using class that implements an efficient closest neighbor search technique in high-dimensional spaces. The class can insert data points, search for neighbors, update neighborhood structures, and find k-nearest neighbors. Explaining the code in detail:

It defines an __init__ method that creates an instance of the HNSW class, having as parameters: data dimension (dim), max element count (max_elements), construction parameters: ef_construction, m, mL, and such attributes as data, layer, links, nodes_num, enter_point.

The _distance method uses the NumPy function np.linalg.norm to calculate the Euclidean distance between two vectors. The _get_random_layer generates a random layer for a data point using a probability function. The _insert method calculates connections and updates information in order to insert a new data point into the HNSW structure.

The _search_layer method searches for a number of neighbors in a layer dependent on distance from a query point. It returns a list of neighbors given a set of layer and parameters. Class methods such as the _select_neighbors_simple and _select_neighbors_heuristic methods optimize the search in picking neighbours. Other methods retrieve and update neighbours at layers of structure, such as the _get_neighborhood_at_layer and _update_neighborhood_at_layer methods, respectively. To maintain the structure's navigability, _add_bidirectional_connections connects a selection of neighboring points.

The fit method inserts some data points into the HNSW structure, and the _k_nn_search method searches for a query vector using k-nearest neighbors. Then, this code section in the if __name__ == "__main__": block initializes an instance of the HNSW class, fits it with randomly produced data points, performs a k-nearest neighbor search for a random query vector, and prints out the nearest neighbors.

The code uses the HNSW method, which efficiently and scales nearest neighbor searches in high-dimensional spaces using a hierarchical and navigable small-world structure.

4. Experiments

We conducted experiments to compare the performance of the HNSW algorithm (with Python bindings of the hnswlib C++ implementation) against FAISS (with IVF-PQ technique) and Annoy. This comparison allows us to assess hybrid (FAISS), tree-based (Annoy), and graph-based (HNSW) techniques together. Key measures of interest were performance in both Euclidean and angular domains, query speed across varied dataset sizes, and building complexity.

Every experiment ran on a MacOS M1 2020, which set certain computing restrictions. Although HNSW may be implemented straight in Python, we decided to use the optimized C++ implementation available via Python bindings. This ensured that the experiments reflected actual performance and were efficient as well.

4.1 Annoy (Tree-Based Approach)

Developed by Eric Bernhardsson in 2015, Annoy (Approximate Nearest Neighbours Oh Yes) is an algorithm, which uses random projections and tree topologies for finding k approximate nearest neighbours. The method can search datasets with up to 1,000 dense dimensions (Bernhardsson, 2015).

4.2 FAISS-IVFPQ (Quantization and Hybrid Approach)

FAISS (Facebook AI Similarity Search), developed by Facebook AI in 2017, is a powerful library for rapid similarity search and clustering of dense vectors. The most well-known strategy from FAISS is the Inverted File with Product Quantification (IVF-PQ) approach. This method employs an inverted index and a quantization-based approach to ensure a balance in accuracy and speed. The inverted index partitions the data space into clusters, whereas product quantization compresses vectors inside each cluster to increase storage and search performance (Johnson, Douze, & Jégou, 2019). (IVF-PQ) is best suited for large-scale datasets with high dimensionality since memory and computational performance are crucial.

4.3 Construction Complexity

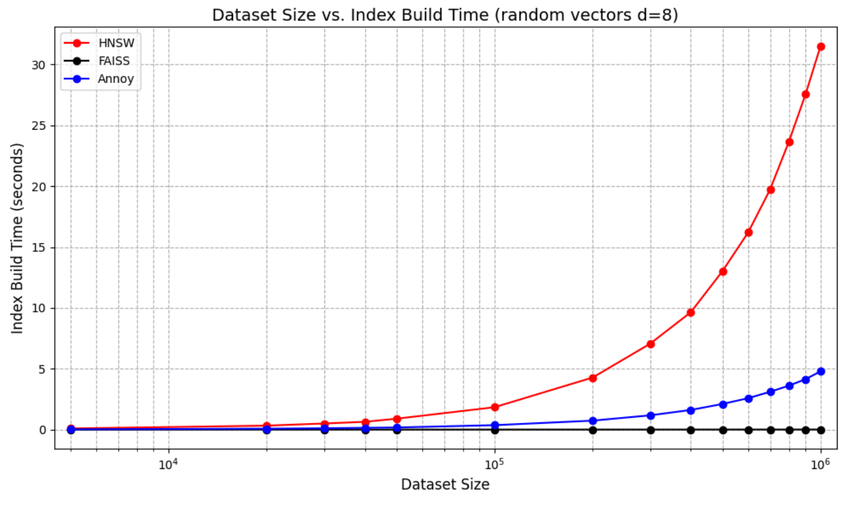

Figure 4. Dataset Size vs. Index Build Time (random vectors d=8)

The time required to generate the index over several dataset sizes measures the construction complexity in our experiment. The *`n_trees`* for annoy were set to 10, *`ef`* for hnsw to 10, and *`nprobe`* for faiss to 10. As seen in Figure 4, the experiment results show notable variations in index construction complexity across the three approaches: Annoy, HNSW, and FAISS (IVF-PQ). To demonstrate scalability, the index build time was assessed over growing dataset sizes (random vectors, d=8).

Indicating its greater computational complexity, HNSW shows the fastest increase in building time as dataset size rises. This is expected as the graph-based approach optimizes search performance by establishing a hierarchical structure with several layers of connections. The building time for big datasets (1.000.000 vectors) exceeds 30 seconds, higher than in the other techniques.

4.4 Speed Comparison by Dataset size

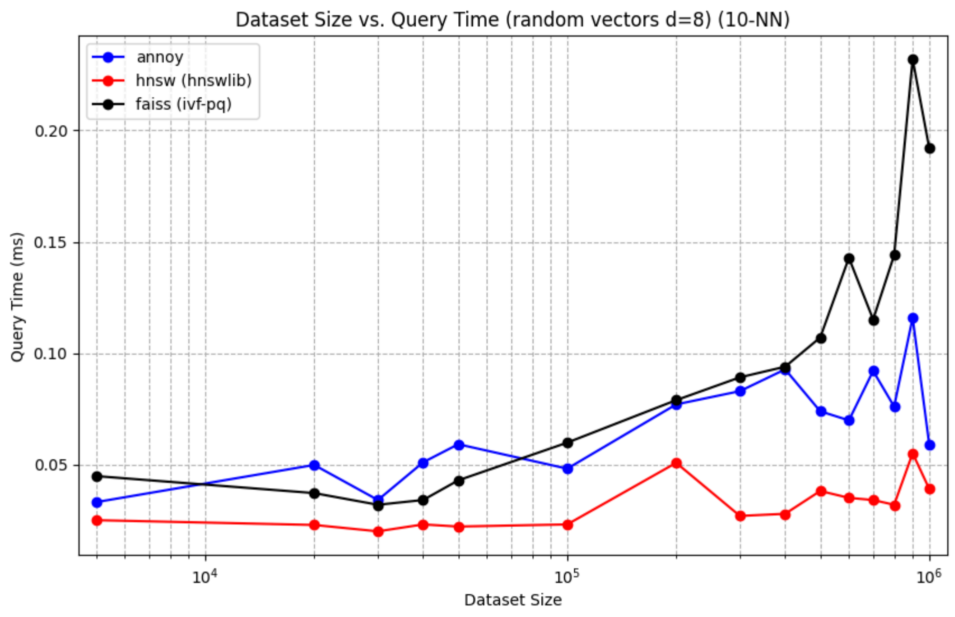

Figure 5. Dataset size vs. Query Time (random vectors d=8) (10-NN)

Random vector datasets (with d=8) were generated to check how effectively the algorithms handled varying dataset sizes in terms of speed (query time). The *`n_trees`* for annoy were set to 10, *`ef`* for hnsw to 10, and *`nprobe`* for faiss to 10. Figure 5 illustrates the query time performance when searching for the 10 nearest neighbors (10-NN) in those datasets. HNSW has the most efficient and consistent query times across all dataset sizes. For the largest dataset with 1,000,000 vectors, the query time was the lowest taking around 0.05ms to retrieve the neighbors. From the results, we might infer that HNSW seems to be the fastest and most scalable technique.

4.5 Comparison in Euclidean Space

Dataset	***Description***	***Size***	***d***
SIFT	Image feature vectors	1M	128
MNIST	Handwritten digit images	60K	100

Table 1. Description of Datasets used for comparison in Euclidean Space (https://github.com/erikbern/ann-benchmarks)

To evaluate HNSW's performance in Euclidean space, we utilized well-known benchmarks for assessing approximate nearest neighbor (ANN) algorithms, including the SIFT and MNIST datasets. 1000 random query vectors were chosen for testing. Table 1 summarizes the key characteristics of these datasets. All the algorithms compared in the experiments support Euclidean distance metrics. The primary metrics for comparison were query time (measured in milliseconds) and recall. Here, recall represents the ratio of correctly retrieved nearest neighbors (true positives) to the total number of actual nearest neighbors in the dataset. For each method, we iteratively changed the hyperparameters to see the overall performance based on the given metrics. Hyperparameters for each algorithm were set optimal as indicated in the respective papers (Malkov & Yashunin, 2018; Johnson et al., 2019; Bernhardsson, 2015).

Hyperparameters set for the experiment:

`ef` (hnsw) : [1, 2, 3, 4, 10, 20, 30, 40, 50, 100, 200, 500]

`n_trees` (annoy) : [1, 2, 3, 4, 10, 20, 30, 40, 50, 100, 200, 500]

`nprobe` (faiss) : [1, 2, 3, 4, 10, 20, 30, 40, 50, 100, 200, 500]

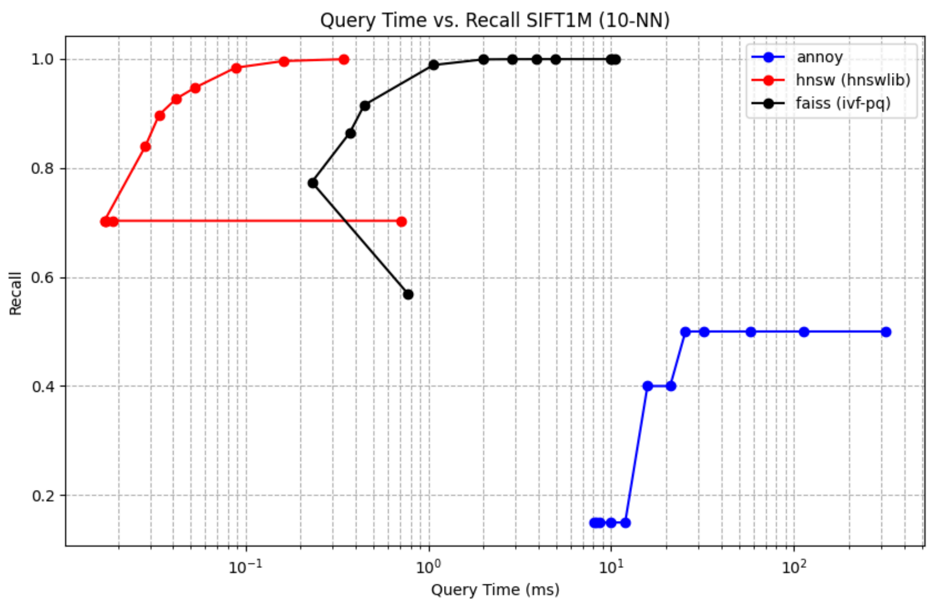

Figure 6. Query time vs. Recall SIFT1M (10-NN)

Figure 6 illustrates the relationship between query time and recall on the SIFT1M dataset. It highlights the trade-offs between accuracy (recall) and retrieval speed, which is critical for approximate nearest-neighbor search. From the experiment conducted, we can conclude that HNSW keeps the fastest query times compared with FAISS and Annoy, while attaining almost perfect recall, which is close to 1.0. Its performance suggests that it offers great efficiency and accuracy making it suitable for tasks where speed and accuracy are required.

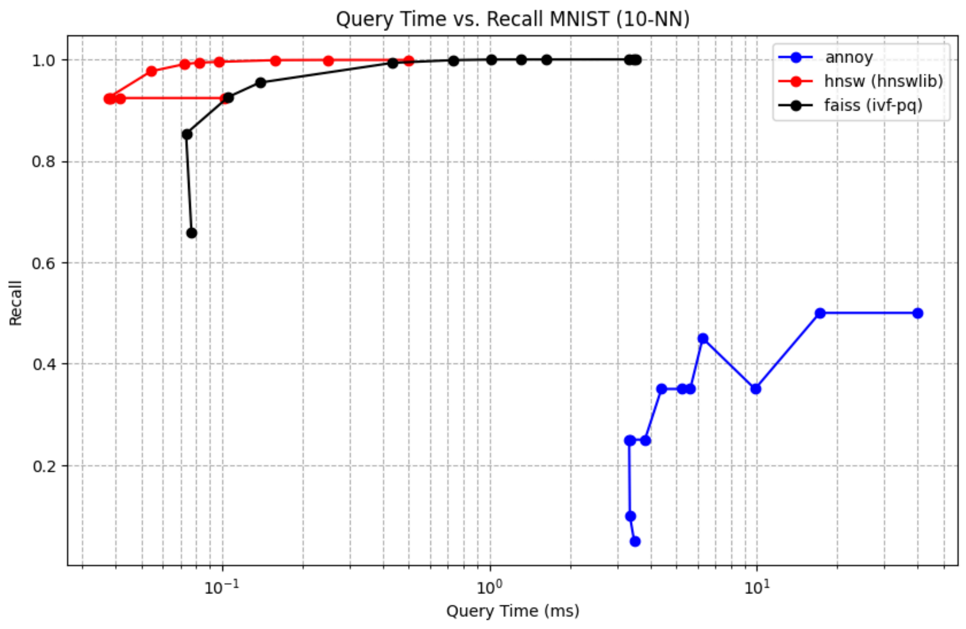

Figure 7. Query Time vs. Recall MNIST (10-NN)

As seen in Figure 7, HNSW showed almost flawless recall with minimum query time for MNIST. FAISS also did well, although with less recall at some points. Annoy, on the other hand, highlighted HNSW's excellence in Euclidean environments since it needed much more time to reach similar recall levels.

4.6 Comparison in Angular space

To evaluate HNSW's performance in Angular space, we utilized GloVe, a well-known benchmark for assessing approximate nearest neighbor (ANN) algorithms. 1000 random query vectors were chosen for testing. Table 2 summarizes the key characteristics of the dataset. To ensure compatibility with the FAISS implementation, we normalized the vectors beforehand to align with the inner product-based similarity. Other two algorithms already had metrics for angular spaces. For each method, we iteratively changed the hyperparameters to see the overall performance based on the given metrics. Hyperparameters for each algorithm were set optimal as indicated in the respective papers (Malkov & Yashunin, 2018; Johnson et al., 2019; Bernhardsson, 2015).

Hyperparameters set for the experiment:

`*ef*` (hnsw) : [1, 2, 3, 4, 10, 20, 30, 40, 50, 100, 200, 500] `*n_trees*` (annoy) : [1, 2, 3, 4, 10, 20, 30, 40, 50, 100, 200, 500]

`*nprobe*` (faiss) : [1, 2, 3, 4, 10, 20, 30, 40, 50, 100, 200, 500]

Dataset	***Description***	***Size***	***d***
GloVe	Word embeddings trained on tweets	1.2M	100

Table 2. Description of Datasets used for comparison in Euclidean Space (https://github.com/erikbern/ann-benchmarks)

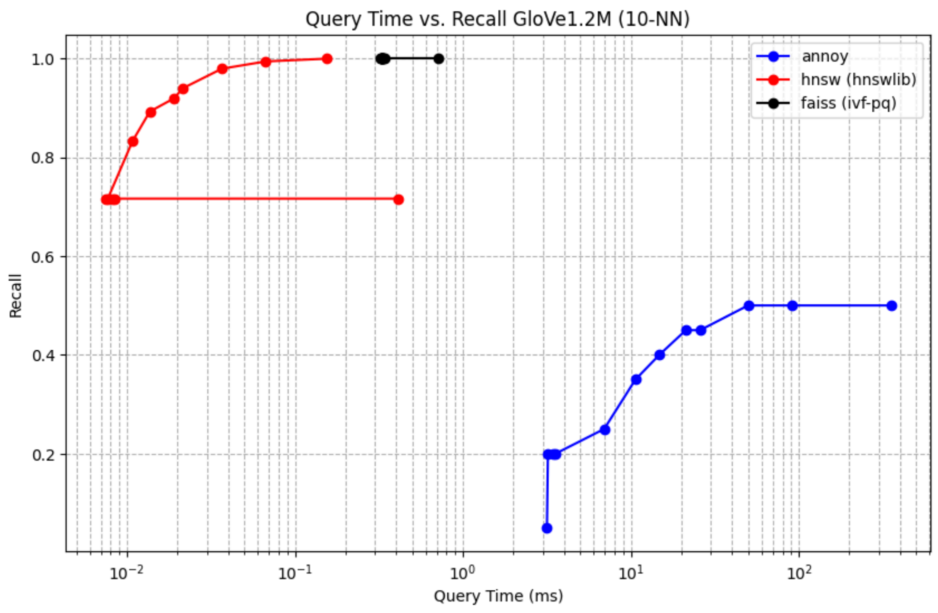

Figure 8. Query Time vs. Recall GloVe1.2M (10-NN)

The results illustrate (Figure 8) that HNSW achieved near-perfect recall at minimal query times, consistently outperforming the other methods in terms of accuracy and speed. FAISS demonstrated a strong balance between recall and query time, particularly in the middle range of the curve, but its performance flattened as query times increased. Annoy, on the other hand, showed slower query times and struggled to reach high recall levels, making it less suitable for this dataset compared to the other algorithms. These results highlight the strengths of HNSW and FAISS in managing large-scale, normalized text embeddings efficiently

5. Conclusion and Improvements

From the experiments conducted, we can show that HNSW performs the best among several approaches and substitutes. HNSW models repeatedly show good performance in terms of query speed, recall, and scalability. Specifically for large datasets, HNSW needed more time to build the graph-based index in the index build time test than FAISS and Annoy; this overhead was offset by its better query efficiency. HNSW proved to be able to scale effectively since it maintained the fastest performance across several dataset sizes for query time. Recall-wise, HNSW routinely outperformed Annoy and FAISS by regularly scoring almost perfect scores across SIFT, MNIST, and GloVe datasets. This highlights HNSW's ability to balance accuracy and speed, making it an excellent choice for high-dimensional nearest-neighbor searches, particularly in Euclidean and Angular spaces.

In the future, we can try to compete HNSW with the newest searching algorithm to better understand the performance of each algorithm and to know which algorithm performs better in every situation.

References

Original Paper: Yu. A. Malkov and D. A. Yashunin, *"Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs,"* arXiv preprint arXiv:1603.09320, 2018. Available: https://arxiv.org/pdf/1603.09320

Miesle, P. (2023, December 6). *Understanding hierarchical navigable small worlds (HNSW)*. DataStax. https://www.datastax.com/guides/hierarchical-navigable-small-worlds

Bernhardsson, E. (2015). *Annoy (Approximate Nearest Neighbors Oh Yeah)*. Retrieved from https://github.com/spotify/annoy

Johnson, J., Douze, M., & Jégou, H. (2019). Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data, 7*(3), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572