CS 6360 - Database Design Project
April 28, 2024
Dr. Jalal Omer
Group 13

Project Phase 5: Final Report

Terrence Li, Stephen George, Rahul Kolla
Vimal Sebastian, Benjamin Ly, Hamzah Dweik

1. INTRODUCTION

Applications requiring information retrieval require fast and memory-efficient solutions to maximize performance. Utilizing the Approximate nearest neighbor search (ANNS), it is possible to find data in a small indexing time complexity. The importance of ANNS lies in its fundamental role in various applications, ranging from recommendation systems and similarly search in large-scale datasets. Nearest neighbor search algorithms play a crucial role in enhancing user experience and optimizing system performance, like identifying similar products in e-commerce platforms and facilitating content-based recommendations in multimedia database systems.

Traditional methods often face the dilemma of striking a balance between memory efficiency and search speed. These methods exhibited theoretical guarantees regarding search time complexity, but they however dealt with high indexing time complexity. The ANNS algorithm is applied to several approaches such as hashing-based, tree-structure, and quantization-based, and the one that outperforms them all: the graph-based approach [1]. Yet, all of them are plagued with scalability and can encounter egregious time complexity issues utilizing the ANNS.

To reduce the search path, this paper will explore a new graph structure named Navigating Spreading-out Graph (NSG) [1] which can outperform the aforementioned implementations. By addressing the inherent limitations of current graph-based methods and introducing innovative techniques to optimize search efficiency and indexing, the paper paves the way for high-performance ANNS solutions. The NSG demonstrates remarkable performance gains over existing approaches by its integration into the billion-scale search engine Taobao, an e-commerce platform under Alibaba Group, and highlights its practical usefulness in handling massive datasets in production-level environments [1].

This paper will corroborate the findings of [1] by implementing the algorithms described and showing their ability to scale with high-dimensional vectors.

2. SURVEY

2.1 The NNS Problem

The Nearest Neighbor Problem (NNS) is the fundamental root of discussion preceding all other concepts referred to in this paper, thus it would be important to state what exactly this problem is before diving into the more in-depth technicalities and implications that the problem has.

We say that the NNS is where [2]:

Given a finite set S of points in space E^d, preprocess S to efficiently return a point $p \in S$ which is closest to a given query point q.

By closest, we refer to the Euclidean distance in this d-dimensional space. If we have a collection of objects for instance that can be characterized by a set of relevant features, where each feature represents dimensions in a high-dimensional attribute space, then we can map these objects as points and find the "most similar" object to the queried object by finding the point that is nearest to the current point queried. For lower dimensional spaces such as d = 2 or 3, this problem is well-solved, however practical applications of similarity searching end up having up to thousands of dimensions, making a simple linear search of points much more difficult and inefficient to compute to find the nearest neighbor. [2]

The NNS problem can be generalized further into the K-Nearest Neighbors Search Problem (kNNs), where we try to find the K nearest points rather than just the nearest point to a query. Due to the large time complexity issues arising from trying to compute the exact closest neighbors to a query in higher-dimensionality spaces, (as most exact-searching algorithms degrade close to linear

searching time – which is inefficient for large datasets) observations have found that it can be sufficient to find approximate neighbors to a query to solve practical use cases. [2] By opting for proximity rather than exactness, current research with experimental results shows that focusing on this presents a favorable tradeoff in loss of accuracy for a much shorter searching time. Therefore current research tries to find the set of points within some distance of a point near the target query. [1]

2.2 Current ANNS Methods

In the realm of ANNS Methods, existing methods can be broadly categorized into non-graph based and graph-based approaches. Non-graph based methods such as KD-trees, R* trees, VA-files, Locality-Sensitive Hashing (LSH), and Product Quantization (PQ) have been widely utilized. [1] However, these methods often encounter challenges, particularly in high-dimensional spaces, due to the curse of dimensionality. This is where, as the dimensionality of the data increases, the number of points that need to be checked also grows exponentially, leading to significant slowdowns in search performance per every extra addition of dimension considered. [1] This limitation stems from the inefficiencies inherent in partitioning and indexing subspaces, ultimately causing non-graph methods to be outperformed by their graph-based counterparts. [1]

Graph-based ANNS methods, on the other hand, leverage the construction of specialized structures to provide time bounds on nearest-neighbor searches. One prevalent approach involves utilizing structures like Delaunay graphs. Although these are theoretically sound, these graphs suffer from the poor performance of high-dimension spaces due to near-complete connectivity, making the number of points needing to be compared for a particular point very high. This renders them impractical for large-scale datasets. [1] Another approach that uses a similar structure is the Relative Neighborhood Graph. These exhibit a lower average out-degree of their points (lower connectivity) but they fall short of meeting the criteria for being considered in the class of Monotonic Search Networks (MSNETs). [1] Transforming graph structures into minimal MSNETs results in prohibitively large indexing complexities, posing practical challenges. [3]

Yet another alternative approach in the graph-based methods of ANNS is called Navigable Small-World Networks (NSW). This paradigm assigns node neighbors and degrees using a probability distribution. While these types of networks demonstrate polyalgorithmic growth in search path length, their build time complexity of $O(n^2)$ proves inadequate for large-scale problems yet again. [1]

Hierarchical NSW graphs (HNSW) aim to approximate both the Delaunay and NSW graphs, offering promising performance. [4] Benchmark comparisons on platforms online indicated that HSNW was one of the most efficient ANNS algorithms.

We also present Randomized Neighborhood Graphs as another avenue for addressing high-dimensional ANNS problems, boasting efficient search times. Their indexing complexities however render them impractical for real-world applications. Moreover, certain algorithms like GNNS, NSW, and DPG require high average-out degrees in their graphs to achieve optimality. While this facilitates effective search operations, it also leads to the creation of excessively large graphs, posing scalability challenges. [4]

The current landscape of ANNS methods encompasses a diverse array of techniques with their own strengths and limitations. Graph-based approaches offer promising avenues for efficient searching since it doesn't suffer as much from the curse of dimensionality as non-graph methods. Challenges still persist in achieving scalability and practicality, which reflects the ongoing pursuit of novel techniques and algorithms such as the NSG that our paper focuses on.

2.3 The Monotonic Relative Neighborhood Graph

The foundation of approximate nearest neighbor search (ANNS) lies in a greedy algorithmic approach, where, at each vertex, the algorithm must explore all adjacent vertices to determine the next best step. [1] Consequently, the complexity of ANNS is heavily influenced by the average out-degree of the graph (the number of neighboring vertices to examine for a particular vertex) and the length of the search path. To address the challenge of scalability, particularly in the context of very large networks comprising billions of nodes, a novel solution emerges in the form of the Monotonic Relative Neighborhood Graph (MRNG), belonging to the class of Monotonic Search Networks (MSNETs). [3].

[1] defines the Monotonic Search Network as:

Given a finite point set S of n points in space E^d , a graph defined on S is a monotonic search network if and only if there exists at least one monotonic path from p to q for any two nodes p, q ∈ S.

[5] proves that the best-first graph search algorithm can detect monotonic paths for an MSNET without the need for backtracking, leading to a time complexity that is equivalent to path length in an MSNET. [1] also observes a proof on MSNETS, which describes a function for the length of monotonic paths that grows extremely slowly, especially for high dimensions, which can be approximated as logarithmic. This implies that the search time complexity of MSNETS is logarithmic, which is an improvement over other methods.

While other MSNETS have very high build time and indexing complexities, MRNGs are inspired by the minimal MSNET building algorithm for RNGs, instead using a less strict selection strategy, resulting in a faster build time [1].

Additionally, MRNGs are sparse, and their maximal degree is constant and independent of the size of the graph [1], solving the high out-degree problem.

Thus, MRNGs provide many improvements over typical MSNETs while maintaining the fast search complexity of its graph family.

2.4 Approximation of MRNGs with Navigating Spreading-Out Graphs

Despite the improvements MRNG has over MSNET, it still has an impractically high indexing time regarding large networks [1] . The Navigating Spreading-Out Graph [1] approximates MRNG while boasting a significantly reduced indexing size compared to MRNG, which addresses the main issue with other graph-based NNS methods.

The steps to build a NSG steps are as follows[1]:

1. Build a kNN graph using the NNDescent Algorithm [6] – a state-of-the-art building method based on experimental benchmarks.
    a. Pick random approximations of kNN for each object.
    b. Iteratively improve the approximation by comparing each object against its current neighbors' neighbors.
    c. Stop when no improvement can be made.
2. Define the Navigating Node, from which all searches begin. This can be found using the centroid as a query and running the Search-on-graph algorithm; the returned nearest neighbor is the medoid, or Navigating Node
3. Generate a candidate neighbor set for each node by performing the Search-on-graph algorithm from the Navigating Node to each other node, adding any intermediate nodes to the candidate set.
4. Span a Depth-First Search Tree on the graph, using the Navigating Node as a root.

By beginning all NSG searches with the Navigating Node, which has a monotonic path to each other node, the NSG mimics how MRNGs have monotonic paths between all nodes. Thus, an NSG search is close to a MRNG search in terms of efficiency [1].

Each node along the path from the Navigating Node to a node *p* is highly likely to be a candidate neighbor due to the monotonicity of the graph [1]. Thus, we can approximate how MRNGs treat all nodes as candidates while avoiding the high time complexity such a strategy entails [1].

The NSG limits the out-degree of each node by disregarding longer edges, though this loses the guaranteed connectivity on the graph [1]. However, by building a DFS tree and connecting any disconnected components as necessary, this reinstates the connectivity of the NSG. Thus, the NSG is sparse because of the out-degree limit and is connected due to the DFS tree. This also saves memory compared to HNSW, which uses a multi-layer graph to remedy high out-degree [1, 7].

[1] indicates the NSG's indexing time provides a major improvement over an MRNG. Additionally, the NSG's search time complexity can be approximated to *O(log n)* [1].

3. TECHNICAL ANALYSIS

This section outlines the two main algorithms implemented from [1]: the Search-on-Graph algorithm, standard for any ANNS problem, and the NSGBuild algorithm, which converts a kNN to a NSG.

3.1 Search-on-Graph

Our Search-on-Graph mirrors the implementation described in [1], where, given a graph G, query node q, and candidate pool size L:

1. Begin with a start node in the neighbor candidate pool *S*
2. Find the first unchecked node *n* in *S* and mark it.

3. For each unchecked node *m* in *G* adjacent to *n*, insert *m* into *S*, sorted based on the ascending order of distance to *q*.
4. If L neighbors in *S* have not been checked, go back to step 2.
5. Return the *k* nearest neighbors of *q* (the first *k* nodes of L)

```
k = 0
while k < L:
    next_k = L

    if returnSet[k].flag:
        returnSet[k].flag = False
        n = returnSet[k].id

        for m in range(0,len(self.final_graph[n])):
            id = self.final_graph[n][m]
            if flags[id]: continue
            flags[id] = True
            dist = self.distance(query,data[id])
            if dist >= returnSet[L-1].distance: continue
            nn = Neighbor(id,dist,True)
            # TODO: make InsertIntoPool
            right = insert_into_pool(returnSet,L,nn)

            if right < next_k:
                next_k = right
    if next_k <= k:
        k = next_k
    else:
        k+=1
for i in range(0,K):
    indices[i] = returnSet[i].id

return indices
```

The Search-on-Graph algorithm. Some preprocessing is omitted for brevity.

From the algorithm description and implementation, it can be observed that the runtime of this algorithm is dependent on the out-degree of nodes in the graph [1]. Hence, by having the maximum out-degree bounded by the structure of the graph, the Search-on-Graph is guaranteed to run more efficiently. In the case of a NSG, the graph is extremely sparse, having a maximum out-degree far smaller than the number of nodes in the graph, which guarantees an approximate *O(log n)* search time.

3.2 Building a NSG

To build a NSG, both a kNN and its dataset are required as well as some build parameters (L, R, and C). The kNN can be attained through external means such as EFANNA.

1. Start by finding the medoid of the dataset by simply finding the centroid vector and treating it as an input to the 'Search-on-Graph' algorithm from Section 3.1 (starting at a random node). The resulting node, *n*, is used as a 'navigating node'
2. For each node, *p*, in the kNN, generate a candidate neighbor set by the following subroutine:
    a. perform Search-on-Graph on *p* with *n* as the starting node.
    b. for each node *q* visited during the search, record the distance and store the node in the candidate set
    c. select at most *m* neighbors from the candidate set, where *m* is the max-out degree of the kNN12
3. A tree is built with *n* as a root using the tree_grow subroutine. While performing a DFS scan, for all nodes that were not present in the scan, link them according to their nearest neighbor using Search_on_Graph

```
def build(self, n, data, parameters):
    nn_graph_path = parameters.get('nn_graph_path')
    range_r = parameters.get('R')
    self.load_nn_graph(nn_graph_path)
    self.data = data
    self.init_graph(parameters)
    cut_graph = [Neighbor() for _ in range(n * range_r)]
    self.link(parameters, cut_graph)
    self.final_graph = [[] for _ in range(n)]

    for i in range(n):
        pool = cut_graph[i * range_r:(i + 1) * range_r]
        pool_size = 0
        for neighbor in pool:
            if neighbor.distance == -1:
                break
            pool_size += 1

        self.final_graph[i] = [neighbor.id for neighbor in pool[:pool_size]]

    self.tree_grow(parameters)

    max_degree = max(len(g) for g in self.final_graph)
    min_degree = min(len(g) for g in self.final_graph)
    avg_degree = sum(len(g) for g in self.final_graph) / n
    self.has_built = True
```

The NSG-Build algorithm.

From the algorithm description and the building implementation, we can see that, as an edge selection strategy over an approximated Delaunay Graph, the runtime of this function will be a slow-growing poly-log polynomial expression [1]. Factoring in the time to build a *k*NN (preprocessing for this algorithm), we see that the overall runtime is determined by the runtime of building the NSG.

4. EVALUATION

The evaluation was conducted by measuring the search time of a query with respect to its candidate neighbor pool size (L) and *k* values. This differs from the testing method used in the NSG paper [1], which

performs a high frequency of queries using a powerful processor. Since we do not have access to such hardware, we settled for this to test individual search times.

4.1 Testing

Each of the following functions was subject to a unit test to ensure correct output using Google Colab, a third-party Python virtual environment:

- Load_data() takes in a kNN graph and outputs a Numpy array with the values contained within the graph
- Load() takes in a NSG and upon successful execution, returns the rows and columns of the NSG
- Search() takes in an object of the IndexNSG class along with a set of queries, performs search, and outputs a search time representing the time it takes to perform the search (Note: this time varies greatly depending on the machine that invokes search. On personal machines the search time is approximately 1-2 seconds. For Google Colab, it was 168 seconds).
- Insert_into_pool() takes a Numpy array populated with Neighbor objects and calculates the inserting index
- build() takes in the size, the data from the kNN graph, and a set of parameters (initialized from the initial invocation: L, R) and constructs an NSG. The output is the max and min degree of the graph and a calculation of the average degree

By testing each of these functions individually, we confirmed the correctness of our algorithms so that we could focus on the interpretation of our data.

4.2 Findings

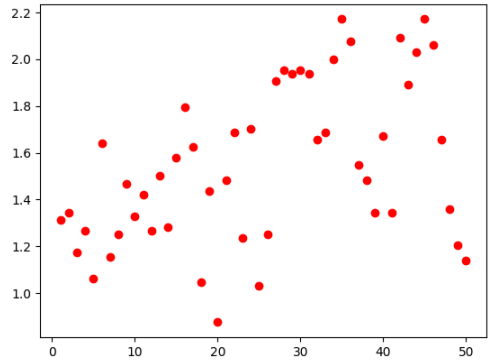
With some variance, we observed the rough logarithmic scaling of search time with k. Additional tests run on repeated searches at L=50 and *k*=40 show that the average time for a search was about 1.5 seconds on Windows and 0.735 seconds on MacOS. There is a much less pronounced correlation in the MacOS experiments comparing the search time with *k*, as the scatterplot shows a very weak to null positive trend when *k* increases. However, as can be observed, runtimes on the Mac were considerably faster with lower deviation from median runtimes, which may be the cause of the visual noise in the graph.

Though we cannot provide a substantial corroboration of the findings in [1] due to technical limitations, we can still observe an indication of high speed of a search on a NSG.

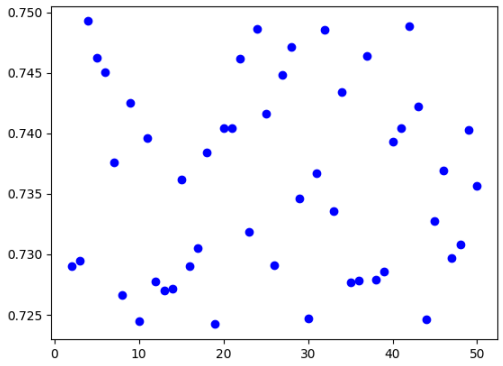
Compared to benchmarks for other ANNS algorithms and accounting for the performance degradation of Python [1, 8], our implementation appears to perform roughly on par or better than other algorithms.

Additional search tests were run with variable L values to observe the effects increased accuracy had on runtime. On both machines, we see a fairly identifiable positive relationship between the search time of a target query's 50 approximate nearest neighbors as a function of the search function's *L* value.

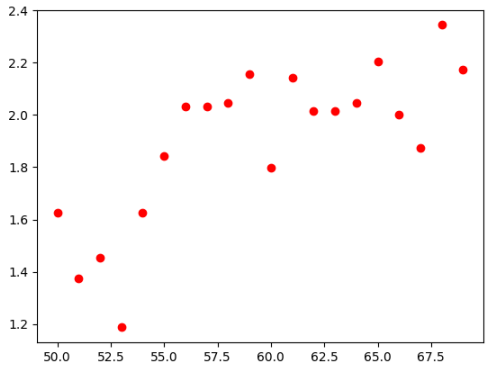
The search time on Windows took about 2.0 seconds while on MacOS was about 0.733 seconds, though the Windows search was generally more consistent



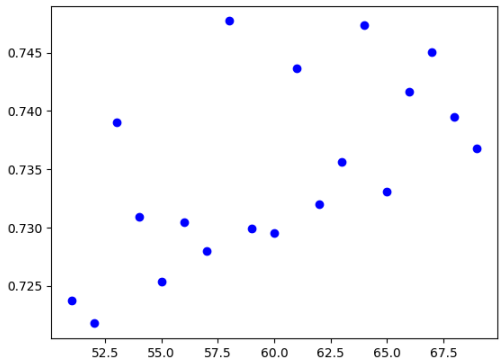
Scatterplot of the search time in seconds (Y-axis) versus the k value (X-axis), with a constant L value of 50 running on a Windows Machine.



Scatterplot of the search time in seconds (Y-axis) versus the *k* value (X-axis), with a constant L value of 50 running on a MacOS Machine.



Scatterplot of the search time in seconds (Y-axis) versus the L value (X-axis), with a constant *k* value of 50 running on a Windows Machine.



Scatterplot of the search time in seconds (Y-axis) versus the L value (X-axis), with a constant *k* value of 50 running on a MacOS Machine.

4.3 Additional Observations from the Project

Issues arose when studying the formatting of input and output vector files that represented the graphs. Particularly, the tool used to create *k*NN graphs, efanna [9], documented its output vector files as bytes of length 4(*k*+2). However, the vectors were actually being written in lengths of 4(*k*+1), causing a byte shift and creating improper *k*NN outputs.

Additionally, when examining the code functionality of efanna, certain header files (e.g. “Parameters.h”) added an additional overhead that hampered readability. The accumulation of these small issues slowed the progress of our implementation, as we had to personally test and confirm the formats and functionalities of items that were documented improperly.

5. SUMMARY

In this paper, we corroborate the performance improvement of NSG over other ANNS methods. We surveyed a brief history of the NNS problem, comparing the performance of different indexing and approximation algorithms. Then, following the criteria set out by [1], we observed whether the trends of our tests matched the analysis of algorithms for NSG. While our findings were limited by processing power, we were able to observe an indication of the running time analysis described by [1].

6. CONTRIBUTIONS

Work in the project was divided into the implementation of the kNN-to-NSG builder and the Search-On-Graph algorithm.

- Hamzah Dweik implemented the IndexNSG class and many of its functions: constructor, save, init_graph, sync_prune, inter_insert, link, and build. He also implemented the Index, Parameters, and Neighbor classes and wrote out the driver file (that included load_data and the code to import and build an NSG from a kNN).

- Stephen George interpreted the original algorithms for NSG. Additionally, he helped implement the Build algorithm and its subroutines by interpreting the pseudocode principles into Python. He also wrote parts of the paper concerning the Build algorithm.

- Benjamin Ly implemented load_nn_graph and get_neighbors. He also gave general support over the algorithms such as load_data and other helper functions for Build by debugging and partially implementing them in Python. He independently verified citations referenced from this report.

- Terrence Li implemented the Search-On-Graph algorithm and surrounding test code. Regarding the paper, he performed preliminary surveys on the background of the Nearest Neighbor problem. Finally, he documented the process of setting up the project from Github and testing it.

- Vimal Sebastian implemented several helper functions for the Search-On-Graph algorithm as well as the kNN-to-NSG builder algorithm. Additionally worked with experimental testing of NSG searches and contributed towards the written survey portion of the paper and finding relevant information of the NSG development.

- Rahul Kolla implemented the Load algorithm and the surrounding test code for NSG implementation. Additionally aided on several helper functions and the subsequent invocation methods. Also served as quality assurance by testing each individual component to ensure smooth performance of the overall algorithms. Contributed to the written paper as well through various sections.

7. SOURCE CODE

The project can be accessed from Github: https://github.com/spevenexe/6360Group13.

The datasets we tested with can be found at http://corpus-texmex.irisa.fr/. We particularly tested with SIFT10K and SIFT1M. The files are quite large and only available through FTP, so "tar" and "wget" or similar options are necessary to download them. More details can be found on Github.

8. REFERENCES

[1] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph," *Proceedings of the VLDB Endowment*, vol. 12, no. 5, pp. 461–474, Jan. 2019. doi:10.14778/3303753.3303754.

[2] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1999, pp. 518–529.

[3] D. W. Dearholt, N. Gonzales, and G. Kurup, "Monotonic Search Networks For Computer Vision Databases," *Twenty-Second Asilomar Conference on Signals, Systems and Computers*, vol. 2, Nov. 1988. doi:10.1109/acssc.1988.754602.

[4] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, Sep. 2014. doi:10.1016/j.is.2013.10.006.

[5] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph," *arXiv.org*, Jul. 01, 2017. https://arxiv.org/abs/1707.00143

[6] Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, Mar. 2016. doi:10.1109/tpami.2018.2889473.

[7] W. Dong, M. Charikar, and K. Li, "Efficient K-nearest Neighbor Graph Construction for Generic Similarity Measures," *Proceedings of the 20th International Conference on World Wide Web*, Mar. 2011. doi:10.1145/1963405.1963487.

[8] ZJULearning, "GitHub - ZJULearning/nsg: Navigating Spreading-out Graph For Approximate Nearest Neighbor Search," GitHub. https://github.com/ZJULearning/nsg (accessed Apr. 28, 2024).

[9] ZJULearning, "GitHub - ZJULearning/efanna_graph: an Extremely Fast Approximate Nearest Neighbor graph construction Algorithm framework," GitHub. https://github.com/ZJULearning/efanna_graph (accessed Apr. 28, 2024).