FINGER: Fast Inference for Graph-based Approximate Nearest Neighbor Search

Patrick H. Chen
UCLA, Los Angeles, CA, USA
patrickchen@g.ucla.edu

Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu
Amazon Search, Palo Alto, CA, USA
{weicheng.cmu,jyunyu.jiang,rofu.yu}@gmail.com

Inderjit S. Dhillon
Google and UT Austin, Austin, TX, USA
inderjit@cs.utexas.edu

Cho-Jui Hsieh
Amazon Search and UCLA, Los Angeles, CA, USA
chohsieh@cs.ucla.edu

ABSTRACT

Approximate K-Nearest Neighbor Search (AKNNS) has now become ubiquitous in modern applications, such as a fast search procedure with two-tower deep learning models. Graph-based methods for AKNNS in particular have received great attention due to their superior performance. These methods rely on greedy graph search to traverse the data points as embedding vectors in a database. Under this greedy search scheme, we make a key observation: many distance computations do not influence search updates so that these computations can be approximated without hurting performance. As a result, we propose FINGER, a fast inference method for efficient graph search in AKNNS. FINGER approximates the distance function by estimating angles between neighboring residual vectors. The approximated distance can be used to bypass unnecessary computations for faster searches. Empirically, when it comes to speeding up the inference of HNSW, which is one of the most popular graph-based AKNNS methods, FINGER significantly outperforms existing acceleration approaches and conventional libraries by 20% to 60% across different benchmark datasets.

KEYWORDS

Approximate K-Nearest Neighbor Search (AKNNS); Similarity Search; Graph-based Approximate K-Nearest Neighbor Search.

1 INTRODUCTION

K-Nearest Neighbor Search (KNNS) is a fundamental problem in machine learning [6], and is applied in various real-world applications in computer vision, natural language processing, and data mining [9, 38, 41]. Further, most of the neural embedding-based retrieval and recommendation algorithms require KNNS in the inference phase to find items that are nearest to a given query [50]. Formally, consider a dataset D with n data points $\{d_1, d_2, ..., d_n\}$, where each data point has m-dimensional features. Given a query $q \in \mathbb{R}^m$, KNNS algorithms return the K closest points in D under a certain distance measure (e.g., $L2$ distance $\|\cdot\|_2$). Despite its simplicity, the cost of finding exact nearest neighbors is linear in the size of a dataset, which can be prohibitive for massive datasets in real-time applications. It is almost impossible to obtain exact K-nearest neighbors without a linear scan of the whole dataset due to a well-known phenomenon called curse of dimensionality [26]. Thus, in practice, an exact KNNS becomes time-consuming or even infeasible for large-scale data. To overcome this problem, researchers resort to Approximate K-Nearest Neighbor Search (AKNNS). An AKNNS method proposes a set of K candidate neighbors $T = \{t_1, \cdots, t_K\}$ to approximate the exact answer. Performance of AKNNS is usually measured by recall@K defined as $\frac{|T \cap A|}{K}$, where A is the set of ground-truth K-nearest neighbors of the query q in the dataset D. Most AKNNS methods try to minimize the search time by leveraging pre-computed data structures while maintaining high recall [27]. Among voluminous AKNNS literature [7, 12, 38, 44, 48], most of the efficient AKNNS methods can be categorized into three categories: quantization methods, space partitioning methods, and graph-based methods. In particular, graph-based methods receive extensive attention from researchers due to their competitive performance. Many papers have reported that graph-based methods are among the most competitive AKNNS methods on various benchmark datasets [3, 7, 17, 44].

Graph-based methods work by constructing an underlying search graph, where each node in the graph corresponds to a data point in D. Given a query q and a current search node c, at each step, an algorithm will only calculate distances between q and all neighboring nodes of c. Once the local search of c is completed, the current search node will be replaced with an unexplored node whose distance is the closest to q among all unexplored nodes. Thus, neighboring edge selection of a data point plays an important role in graph-based methods as it controls the complexity of the search space. Consequently, most recent research is focused on how to construct different search graphs or design heuristics to prune edges in a graph to achieve efficient searches [17, 27, 35, 43?]. Despite different methods having their own advantages, there is no clear winner among these graph construction approaches on all datasets. Following a recent systematic evaluation protocol [3], we evaluate performance by comparing throughput versus recall@10
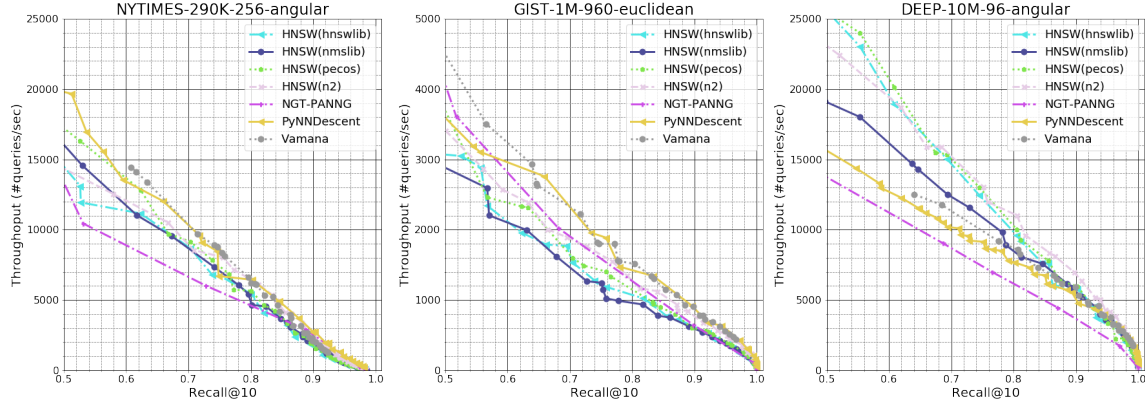
Figure 1: Comparison of state-of-the-art graph-based libraries on three benchmark datasets. Throughput versus recall@10 curve is used as the metric, where a larger area under the curve corresponds to a better method. We can observe no single method outperforms the rest on all datasets. Best viewed in color.

curves, where a larger area under the curve corresponds to a better method. As shown in Figure 1, many graph-based methods achieve similar performance on three benchmark datasets. A method (e.g., PyNNDescent [13]) can be competitive on a dataset (e.g., GIST-1M-960) while another method (e.g., HNSW [35]) performs better on the other dataset (e.g., DEEP-10M-96). These results suggest there might not be a single graph construction method that works best, which motivates us to consider the research question: *Other than improving an underlying search graph, is there any other strategy to improve search efficiency of all graph-based methods?*.

In this paper, instead of proposing yet another graph construction method, we show that for a given graph, part of the computations in the inference phase can be substantially reduced. Specifically, we observe that after a few node updates, most of the distance computations will not influence the search update. This suggests the complexity of distance calculation during an intermediate stage can be reduced without hurting performance. Based on this observation, we propose FINGER, **F**ast **IN**ference for **G**raph-based approximated nearest neighbor s**EaR**ch, which reduces computational cost in a graph search while maintaining high recall. Our contribution are summarized as follows:

- We provide an empirical observation that most of the distance computations in the prevalent best-first-search graph search scheme do not affect final search results. Thus, we can reduce the computational complexity of many distance functions.
- Leveraging this characteristic, we propose an approximated distance based on modeling angles between neighboring vectors. Unlike previous methods which directly approximate whole L2-distance or inner-product, we propose a simple yet effective decomposition of the distance function and reduce the approximation error by modeling only the angle between residual vectors. This decomposition yields a much smaller approximate error and thus a better search result.
- We provide an open source efficient C++ implementation of the proposed algorithm FINGER on the popular HNSW graph-based method. HNSW-FINGER outperforms many popular graph-based AKNNS algorithms in wall-clock time across various benchmark datasets by 20% to 60%.

2 RELATED WORK

There are three major directions in developing efficient approximate K-Nearest-Neighbours Search (AKNNS) methods. The first direction traverses all elements in a database but reduce the complexity of each distance calculation; quantization methods represent this direction. The second direction partitions the search space into regions and only search data points falling into matched regions, including tree-based methods [42] and hashing-based methods [8]. The third direction is graph-based methods which construct a search graph and convert the search procedure into a graph traversal.

2.1 Quantization Methods

Quantization methods compress data points and represent them as short codes. Compressed codes consume less storage and thus achieve more efficient memory bandwidth usage [22]. In addition, the complexity of distance computations can be reduced by computing approximate distances with the pre-computed lookup tables. Quantization can be done by random projections [34], or learned by exploiting structure in the data distribution [36, 39]. In particular, the seminal Product Quantization method [28] separates the data feature space into different parts and constructs a quantization codebook for each chunk. Product Quantization has become the cornerstone for most recent quantization methods [14, 22, 37, 46]. There is also work focusing on learning transformations in accordance with product quantization [19]. Most recent quantization methods achieve competitive results on various benchmarks [22, 30].

2.2 Space Partition Methods

Hashing-based Methods generate low-bit codes for high dimensional data and try to preserve the similarity among the original distance measure. Locality sensitive hashing [20] is a representative framework that enables users to design a set of hashing functions. Some data-dependent hashing functions have also been designed [25, 45]. Nevertheless, a recent review [7] reported the simplest random-projection hashing [8] actually achieves the best performance. According to this review, the advantage of hashing-based methods is simplicity and low memory usage; however, they are significantly outperformed by graph-based methods.

Tree-based Methods learn a recursive space partition function as a tree following some criteria. When a new query comes, the learned partition tree is applied to the query and the distance computation is performed only on relevant elements falling in the same sub-tree. Representative methods are KD-tree forest with a unified search queue [42] and R^*-tree [5]. Previous studies observed that tree-based methods only work for very low-dimensional data and their performances drop significantly for high-dimensional problems [7].

2.3 Graph-based Methods

Graph-based methods date back to theoretical work in the graph theory of graph paradigm with proper theoretical properties [4, 11, 32]. However, these theoretical guarantees only work for low-dimensional data [4, 32] or require expensive ($O(n^2)$ or higher) index building complexity [11], which is not scalable to large-scale datasets. Recent studies are mostly geared toward approximations of different proximity graph structures in order to improve approximate nearest neighbor search. An early work showing the practical value of these methods could be found in [2], and is a series of works on approximating K-nearest-neighbour graphs [16, 23, 24, 29]. Most recent studies approximate monotonic graphs [18] or relative neighbour graph [2, 35]. In essence, these methods first construct an approximated K-nearest-neighbour graph and prune redundant edges by different criteria inspired by different proximity graph structures. Some other works mixed the above criteria with other heuristics to prune the graph [17, 27]. Some pruning strategies can even work on randomly initialized dense graphs [27]. According to various empirical studies [3, 7, 24], graph-based methods achieve very competitive performance among all AKNNS methods. Despite concerns about scalability of graph-based methods due to their larger memory usage [14], it has been shown that graph-based methods can be deployed in billion scale commercial usage [18]. In addition, recent studies also demonstrated that graph-based AKNNS can scale quite well on billion-scale benchmarks when implemented on SSD hard-disks [10, 27].

In this work, we aim at demonstrating a generic method to accelerate the inference speed of graph-based methods so we will mainly focus on in-memory scenarios. There are also prior works working on better search schemes on graph-based methods by using KD-Tree [40] and clustering [47]. We will provide more details and compare to these baseline methods in Section 4.

3 FINGER: FAST INFERENCE FOR GRAPH-BASED AKNNS

In this section, we first provide a motivating observation suggesting that approximating distance computations can accelerate inference of graph-based methods. Next, we analyze the distance computation in a graph search and figure out that the key to approximate the distance is to estimate angles between neighboring residual vectors. We then propose FINGER, **F**ast **IN**ference for **G**raph-based approximated nearest neighbor s**EaR**ch, a low-rank estimation method plus a distribution matching technique to improve the inference speed of general graph-based algorithms.

3.1 Observation: Most distance computations do not contribute to better search results

Once a search graph is built, graph-based methods use a greedy-search strategy (Algorithm 1) to find relevant elements of a query in a database. It maintains two priority queues: candidate queue that stores potential candidates to expand and top results queue that stores current most similar candidates (line 1). At each iteration, it finds the current nearest point in the candidate queue and explores its neighboring points. An upper-bound variable records the distance of the farthest element from the current top results queue to the query q (line 4). The search will stop when the current nearest distance from the candidate queue is larger than the upper-bound (line 5), or there is no element left in the candidate queue (line 2). The upper-bound not only controls termination of the search but also determines if a point will be present in the candidate queue (line 11). An exploring point will not be added into the candidate queue if the distance from the point to the query is larger than the upper-bound.

The upper-bound plays an important role as we need to spend computational resources on distance calculation ($dist()$ function in line 11) but it might not influence search results if the distance is larger than the upper-bound. Empirically, as shown in Figure 2, we observe in two benchmark datasets that most explorations of graph search end up having larger distances than the upper-bound. Especially, starting from the mid-phase of a search, ==over 80 % of distance calculations are larger than the upper-bound==. Using greedy graph search will inevitably waste a significant amount of computing time on non-influential operations. [33] also found this phenomenon and proposed to learn an early termination criterion by an ML model. Instead of only focusing on the near-termination phase, we propose a more general framework by incorporating the idea of reducing the complexity of distance calculations into a graph search. The fact that most distance computations do not influence search results suggests that we don't need to have exact distance computations. A faster distance approximation can be applied in the search.

3.2 Modeling Distributions of Neighboring Residual Angles

In this section, we will derive an efficient method to approximate the distance calculations in the search. While most existing methods approximate the whole L2-distance or inner-product directly, instead, in this section we will show that by simple manipulations, we only need to model angles between neighboring residual pairs and thus a much small approximation error could be achieved. Given a query q and the current point c that is nearest to the query in the candidate queue, we will expand the search by exploring neighbors of c in Line 7 of Algorithm 1. Consider a specific neighbor of c called d, we have to compute distance between q and d in order to update the search results. Here, we will focus on the $L2$ distance (i.e., $Dist = \|q - d\|_2$). The derivations of inner-product and angle distance are provided in Appendix D. As shown in the previous section, most distance computations will not contribute to the search in later stages, we aim at finding a fast approximation of $L2$ distance. ==A key idea== is that we can leverage c to represent q (and d) as a vector along c (i.e., projection) and a vector orthogonal

(a) FashionMNIST-60K-784
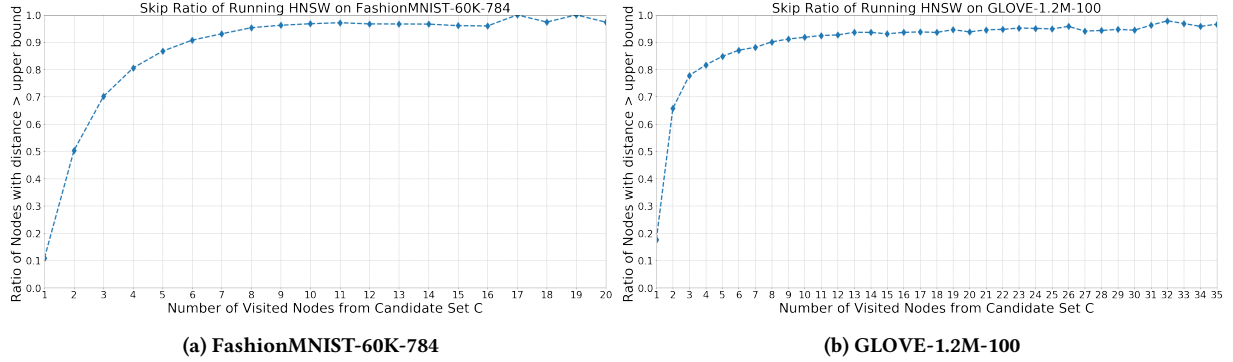
(b) GLOVE-1.2M-100

Figure 2: Empirical observation that distances between query and most points in a database are larger than the upper-bound. (a) shows results on FashionMNIST-60K-784 dataset and (b) shows results on Glove-1.2M-100 dataset. We observed that starting from the 5th step of greedy graph search (i.e., running line 2 in Algorithm 1 five times), both experiments show more than 80% of data points will be larger than the current upper-bound. These distance computations won't affect search updates.

Algorithm 1: Greedy Graph Search

Input: graph G, query q, start point p, distance $dist()$,
  number of nearest points to return efs
Output: top results queue T

1 candidate queue $C = \{p\}$, currently top results queue
  $T = \{p\}$, visited set $V = \{p\}$;
2 **while** C *is not empty* **do**
3     $cur \leftarrow$ nearest element from C to q (i.e., current nearest
  point to expand) ;
4     $ub \leftarrow$ distance of farthest element from T to q (i.e.,
  upper bound of the candidate search) ;
5   **if** $dist(cur, q) > ub$ **then**
6   return T
7   **for** *point* $n \in neighbour$ *of cur in* G **do**
8   **if** $n \in V$ **then**
9   continue
10   V.add(n)
11   **if** $dist(n, q) \leq ub$ *or* $|T| \leq efs$ **then**
12   C.add(n)
13   T.add(n)
14   **if** $|T| > efs$ **then**
15   remove farthest point from T to q
16          $ub \leftarrow$ distance of farthest element from T to q
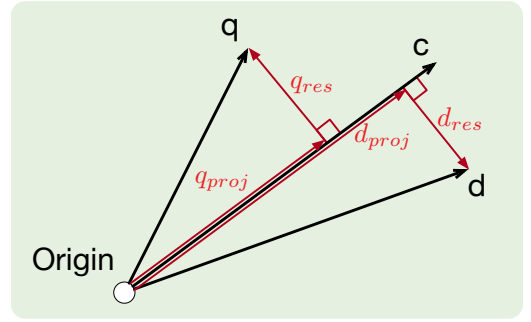 (i.e., update ub)
17 return T

---

to c (i.e., residual):

$$q = q_{proj} + q_{res}, \quad q_{proj} = \frac{c^T q}{c^T c} c, \quad q_{res} = q - q_{proj}. \quad (1)$$

A schematic illustration of this decomposition is shown in Figure 3. In other words, we treat each center node as a basis and project the query and its neighboring points onto the center vector so query and data can be written as $q = q_{proj} + q_{res}$ and $d = d_{proj} + d_{res}$

Figure 3: Decomposition by center point. Query and neighboring data point can be expressed by vectors parallel and orthogonal to the center vector. We named the parallel vector "proj" (projection) and the orthogonal vector "res" (residual).

respectively. To this end, the squared $L2$ distance can be written as:

$$
\begin{aligned}
Dist^2 &= \|q - d\|_2^2 = \|q_{proj} + q_{res} - d_{proj} - d_{res}\|_2^2 \\
&= \|(q_{proj} - d_{proj}) + (q_{res} - d_{res})\|_2^2 \\
&= \|(q_{proj} - d_{proj})\|_2^2 + \|(q_{res} - d_{res})\|_2^2 \\
&\quad + 2(q_{proj} - d_{proj})^T (q_{res} - d_{res}) \\
&\overset{(a)}{=} \|(q_{proj} - d_{proj})\|_2^2 + \|(q_{res} - d_{res})\|_2^2 \\
&= \|(q_{proj} - d_{proj})\|_2^2 + \|q_{res}\|_2^2 + \|d_{res}\|_2^2 - 2q_{res}^T d_{res}, \quad (2)
\end{aligned}
$$

where (a) comes from the fact that projection vectors are orthogonal to residual vectors so the inner product vanishes. For d_{proj} and d_{res}, we can pre-calculate these values after the search graph is constructed. For q_{proj}, notice that center node c is extracted from the candidate queue (Line 3 of Algorithm 1). That means we must have already visited c before. Thus, $\|q - c\|_2$ has been calculated and we can get $q^T c$ by a simple algebraic manipulation:

$$q^T c = \frac{\|q\|_2^2 + \|c\|_2^2 - \|q - c\|_2^2}{2}.$$

Since calculation of $\|q\|_2^2$ is a one-time task for a query, it's not too costly when a dataset is moderately large. $\|c\|_2^2$ can again be pre-computed in advance so $q^T c$ and thus q_{proj} can be obtained in just a few arithmetic operations. Also notice that $\|q\|_2^2 = \|q_{proj}\|_2^2 + \|q_{res}\|_2^2$ as q_{proj} and q_{res} are orthogonal, so we can get $\|q_{res}\|_2^2$ by calculating $\|q\|_2^2 - \|q_{proj}\|_2^2$ in few operations too.

After the above manipulation, the only uncertain term in Eq. (2) is $q_{res}^T d_{res}$. If we can estimate this term with less computational resources, we can obtain a fast yet accurate approximation of $L2$ distance. Since we have no direct access to the distribution of q and thus q_{res}, we hypothesize we can instead use the distribution of residual vectors between neighbors of c to approximate the distribution of $q_{res}^T d_{res}$ term. The rationale behind this is as we only approximate $q_{res}^T d_{res}$ when q and c are close enough (i.e., c is selected in Line 3 of Algorithm 1), both q and d could be treated as near points in our search graph and thus interaction between q_{res} and d_{res} might be well approximated by $d_{res}'^T d_{res}$, where d' is another neighbouring point of c and d_{res}' is its residual vector. Formally, given an existing search graph $G = (D, E)$, where D are nodes in the graph corresponding to data points and E are edges connecting data points, we collect all residual vectors into $D_{res} \in \mathbb{R}^{m \times |E|}$, where $|E|$ is total number of edges in G. We assume D_{res} spans the whole space which residual vectors lie in. Obtaining approximated distance of residual vectors can then be formulated as the following optimization problem:

$$\arg\min_{P \in \mathbb{R}^{r \times m}} \mathbb{E}_{x, y \sim D_{res}} \left\| \|Px - Py\|_2^2 - \|x - y\|_2^2 \right\|_2, \quad (3)$$

where we aim at finding an optimal projection matrix P minimizing the approximating error over the residual pairs D_{res} from training data. It is not hard to see that the Singular Value Decomposition (SVD) of D_{res} will provide an answer to the above optimization problem. Nevertheless, low-rank approximation would not be practical as it consumes much more memory usage. Since we have to save low-rank coordinates for residual vector of each edge, total additional memory is $r \times |E| \times 4$ bytes, where 4 comes from using 32 bits floating points to save each coordinate. For a million scale dataset, with a small rank $r = 16$ and a moderately complex graph (i.e., $|E| \approx 5e7$, it will still cost additional 3.2 GB to save and operate. This greatly inhibits the potential deployment on larger datasets and we need to seek a more memory-efficient approach to estimate $q_{res}^T d_{res}$.

An intuitive idea to reduce the memory is not to save full floating point precision. We can use IEEE FP16 [1] or even self-defined precision [31] to reduce memory consumption. Following this idea, the extreme case is to just use 1 bit to store the sign of the result, and this connects to the canonical theory in Locality Sensitive Hashing (LSH) [8]. Specifically, Random Projection Locality Sensitive Hashing (RPLSH) samples r random vectors from Normal distributions to form a hashing basis. The angles between vectors can be estimated by the following lemma.

LEMMA 1 (LEMMA 3.2 IN [21]). *Given r random vectors $B = \{v_i\}_{i=1}^r$ sampled from a Gaussian Distribution, the estimate for the angle between vectors x and y is given by*

$$\frac{1}{\pi r} \sum_i sgn(x^T v_i) \neq sgn(y^T v_i).$$

Algorithm 2: FINGER Graph Search

Input: graph G, query q, starting point p, learend RPLSH basis B, distance function dist(), approximate distance appx(), pre-calculated information S, number of nearest points to return efs

Output: top candidate set T

1   Query Projection result $Y = q^T B$
2   candidate set $C = \{p\}$
3   dynamic list of currently best candidates $T = \{p\}$
4   visited $V = \{p\}$
5   **while** C *is not empty* **do**
6   cur $\leftarrow$ nearest element from C to
7   ub $\leftarrow$ distance of the farthest element from T to q (i.e., upper bound of the candidate search)
8   **if** *dist(cur, q) > ub* **then**
9   return T
10   **for** *point $n \in$ neighbour of cur in G* **do**
11   **if** *$n \in V$* **then**
12   continue
13   V.add(n)
14   **if** *#updates of cur > 5 times* **then**
15   e = appx(n,q,S,Y) // Approximate Eq.(2)
16   **else**
17   e = dist(n, q) // exact distance Eq.(2)
18   **if** *$e \leq ub$ or $|T| \leq efs$* **then**
19   update distance to be dist(n,q)
20   C.add(n)
21   T.add(n)
22   **if** *$|T| > efs$* **then**
23   remove farthest point to q from T
24   ub $\leftarrow$ distance of the farthest element from T to q (i.e., update ub)

25   return T

Although this approximation cannot achieve optimal value of the above optimization problem, it uses much less memory as the low-rank results are now stored in binary representation instead of full 32 bits precision. For example, when $r = 8$ it only takes 1 byte to save the pre-computed results, which is much smaller than $8 \times 4 = 32$ bytes used by low-rank based approximations. To leverage the idea of RPLSH in our approximation, we need to make two adjustments. First, notice that above lemma is used to estimate angles between vectors whereas we want to estimate inner-product. Thus, we have to further decompose $q_{res}^T d_{res}$ into $\|q_{res}\|\|d_{res}\|\cos(q_{res}, d_{res})$ and calculate hamming distance between signed binarized result to estimate only $\cos(q_{res}, d_{res})$. Consequently, $\|d_{res}\|$ needs to be pre-compute and stored. Second, vanilla random projection guarantees worst case performance [15] and it is oblivious of the data distribution. Since we can sample abundant neighboring residual vectors from the training database, we can leverage the data information to obtain a better approximation. Instead of generating random Gaussian basis, we used top eigenvectors learned from residual pairs D_{res} which will better capture the span of residual vectors.

We will show in Section 4 that this modification achieves better results than using random projections. By using signed LSH to store the low-precision low-rank result, we could greatly reduce the memory usage. Detailed analysis of memory usage and a case study is shown in Appendix C.

3.3 Overall Algorithm of FINGER

Algorithm 2 summarizes how FINGER works. Our aim is to provide a generic acceleration for all graph-based search. Thus, FINGER can applied on top of any existing graph G. Given a query, FINGER firstly compute query basis multiplications (line 1 in Algorithm 2). This is a one-time computation so the cost is negligible when dataset is moderately large. As we mentioned in Section 3.1, most exact distance computation will not lead to an update of candidate set after expanding 5 times of candidate set; therefore, in Line 14 of Algorithm 2, FINGER uses exact distance when exploring first 5 candidates from the candidate set, and starting from the 6th iteration, FINGER uses approximation distance to scan. Approximation function takes neighboring node, query node, query projections and more pre-computed and stored information S. We leave the details of how pre-computed information is used to obtain approximate distance and time complexity analysis in Appendix B.

4 EXPERIMENTS

Baseline Methods. We compare FINGER to the most competitive graph-based and quantization methods. We include different implementations of the popular HNSW methods, such as NM-SLIB [35], n2[1], PECOS [49] and HNSWLIB [35]. We also compare other graph construction methods include NGT-PANNG [43] , VAMANA(DiskANN) [27] and PyNNDescent [13]. Since our goal is to demonstrate FINGER can improve search efficiency of an underlying graph, we mainly include these competitive methods with good python interface and documentation. For quantization methods, we compare to the best performing ScaNN [22] and Faiss-IVFPQFS [30]. In experiments, we combine FINGER with HNSW as it is a simple and prevalent method. The implementation of HNSW-FINGER is based on a modification of PECOS as its codebase is easy to read and extend. Pre-processing cost is discussed in Appendix C.

Evaluation Protocol and Dataset. We follow the latest ANN-benchmark protocol [3] to conduct all experiments. Instead of using a single set of hyperparameter, the protocol searches over a predefined set of hyper-parameters[2] for each method, and reports the best performance over each recall regime. In other words, it allows methods to compete others with its own best hyper-parmameters within each recall regime. We follow this protocol to measure recall@10 values and report the best performance over 10 runs. Results will be presented as throughput versus recall@10 charts. A method is better if the area under curve is larger in the plot. All experiments are run on AWS r5dn.24xlarge instance with Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz. We evaluate results over both $L2$-based and angular-based metric. We represent a dataset with the following format: (dataset name)-(training data size)-(dimensionality of dataset). For $L2$ distance measure, we evaluate on FashionMNIST-60K-784, SIFT-1M-128, and GIST-1M-960.

For cosine distance measure, we evaluate on NYTIMES-290K-256, GLOVE-1.2M-100 and DEEP-10M-96. More details of each dataset can be found in [3]. For search hyper-parameters, we follow the same set of search grid as hnmslib used in ann-benchmark repository. In addition, we search over $r = 64$ and 128 number of basis.

4.1 Improvements of FINGER over HNSW

In Figure 4, we demonstrate how FINGER accelerates the competitive HNSW algorithm on all datasets. Since FINGER is implemented on top of PECOS, it is important for us to check if PECOS provides any advantage over other HNSW libraries. Results verify that across all 6 datasets, the performance of PECOS does not give an edge over other HNSW implementations, so the performance difference between FINGER and other HNSW implementations could be mostly attributed to the proposed approximate distance search scheme. We observe that FINGER greatly boosts the performance over all different datasets and outperforms existing graph-based algorithms. FINGER works better not only on datasets with large dimensionality such as FashionMNIST-60K-784 and GIST-1M-960, but also works for dimensionality within range between 96 to 128. This shows that FINGER can accelerate the distance computation across different dimensionalities. Results of comparison to most competitive graph-based methods are shown in Figure 7 of Appendix A. Briefly speaking, HNSW-FINGER outperforms most state-of-the-art graph-based methods except FashionMNIST-60K-784 where PyNNDescent achieves the best and HNSW-FINGER is the runner-up. Notice that FINGER could also be implemented over other graph structures including PyNNDescent. We chose to build on top of HNSW algorithm only due to its simplicity and popularity. Studying which graph-based method benefits most from FINGER is an interesting future direction. Here, we aim at empirically demonstrating approximated distance function can be integrated into the greedy search for graph-based methods to achieve a better performance.
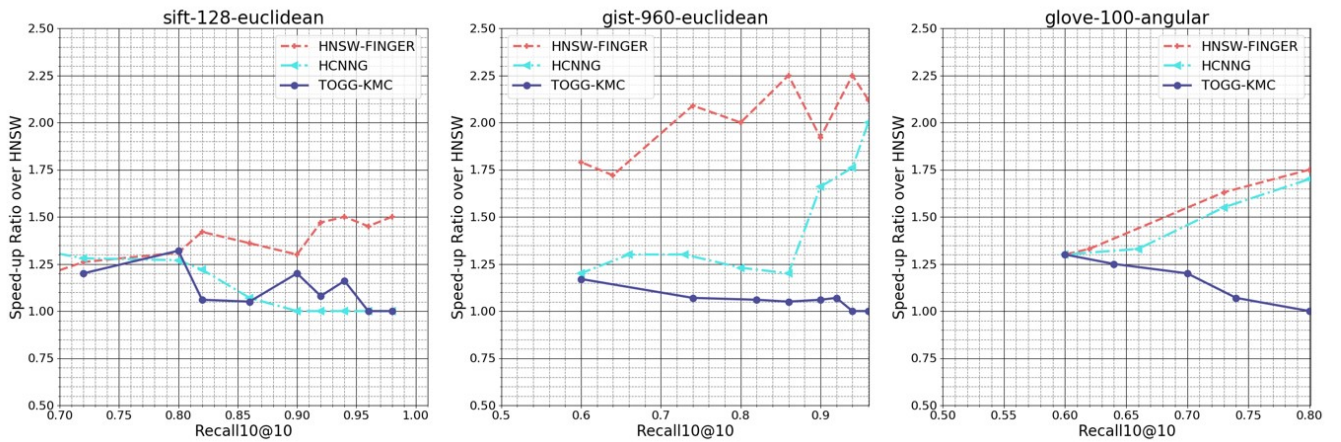
4.2 Comparison to Previous Search Methods

As noted in Section 2, Xu et al. [47] and Munoz et al. [40] also propose better search methods over vanilla greedy algorithms. In sum, TOGG-KMC [47] uses KD-Tree or clustering to select querying neighbor points, and add a fine-tuned step when searching points near the query. HCNNG [40] uses KD-tree to select points in the same direction as the query. Notice that both TOGG-KMC and HCNNG methods select a subset of points to query but still use full exact distance. Whereas, FINGER still explores all neighbors and use a faster yet accurate enough approximation to scan the distances. Since HCNNG did not release code, we could only use reported results as in Fig. 7 of [40]. Munoz et al. [40] only reported speedup-ratio over exact nearest neighbor search, so we cannot directly compare the throughput numbers. Instead, in this section we will report its speedup ratio over HNSW graph with greedy search algorithm. For TOGG-KMC, we run the released code[3] on greedy (GA) and proposed method (TOGG-KMC) setup and compute the speed-up ratio of TOGG-KMC over GA. Munoz et al. [40] only includes results on SIFT-1M-128, GIST-1M-960, and GLOVE-1.2M-100 datasets so we could only compare speedup ratios on these datasets, and the result is shown in Figure 5. As we can observe
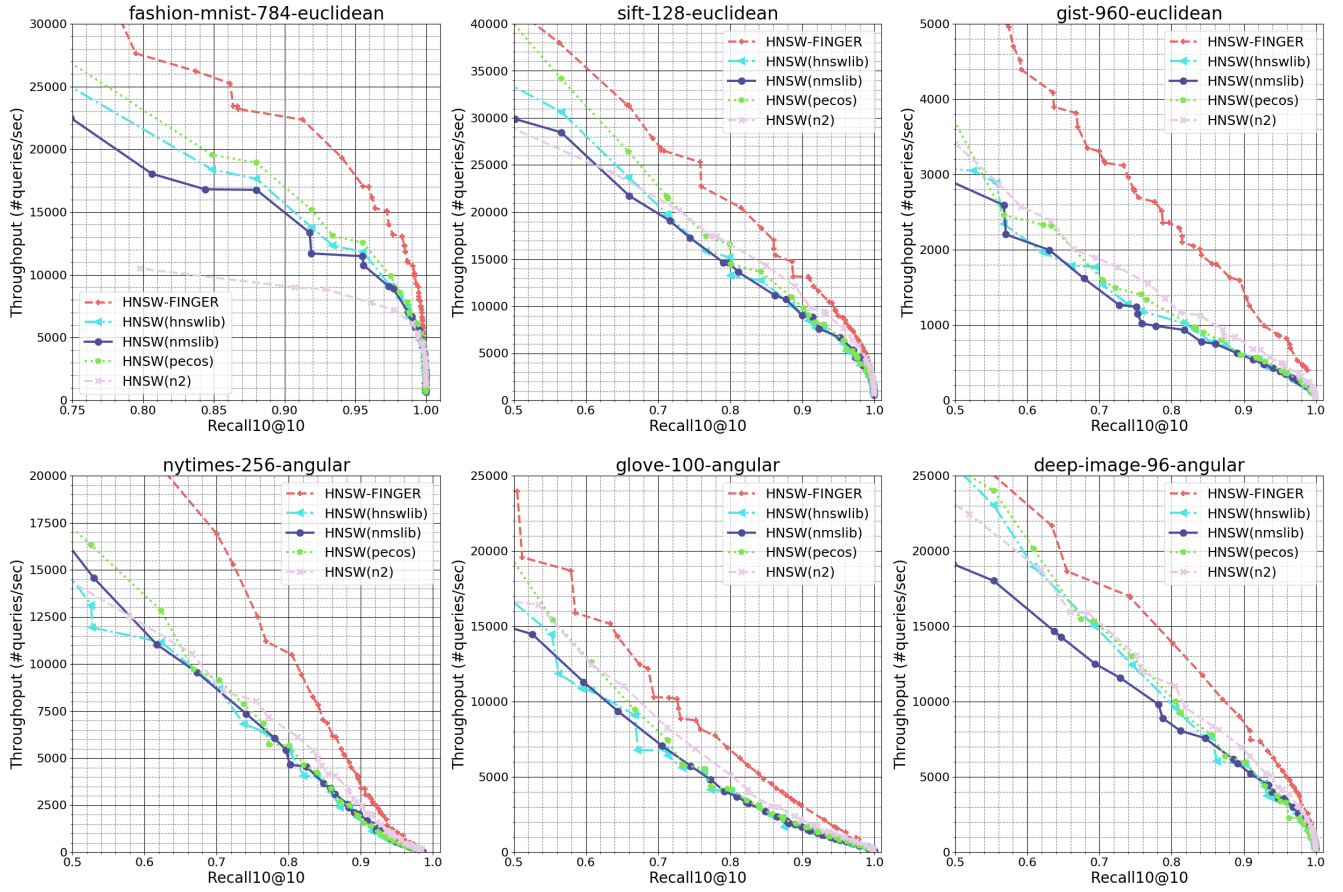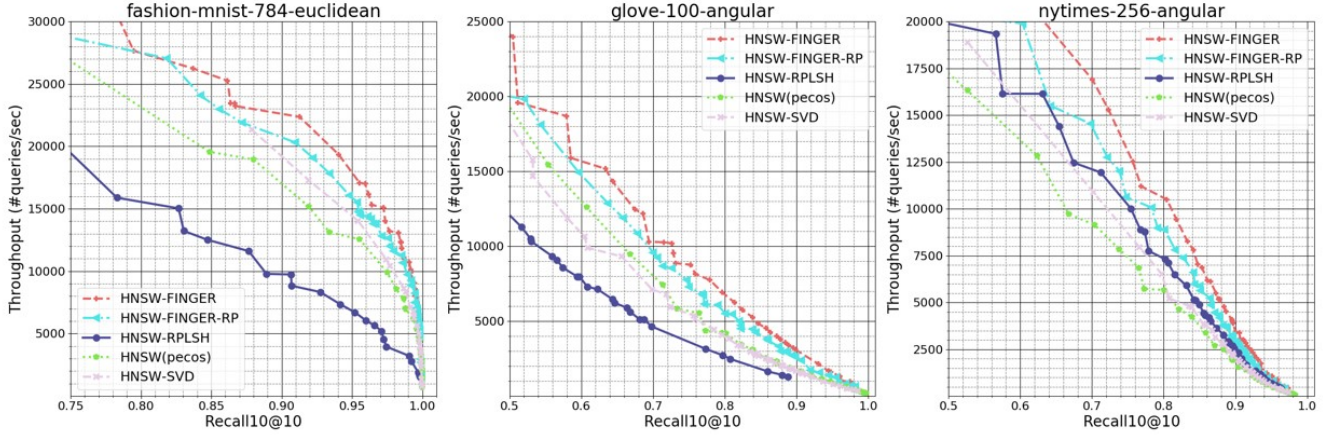
Figure 4: Experimental results of HNSW graph-based methods. Throughput versus Recall@10 chart is plotted for all datasets. Top row presents datasets with $L2$ distance measure and bottom row presents datasets with angular distance measure. We can observe a significant performance gain of FINGER over all existing HNSW graph-based implementations. Best viewed in color.

Figure 5: Experimental results of comparisons to previous methods. X-axis denotes the recall@10 values. Y-axis denotes the speed-up of each algorithm over HNSW with greedy search algorithm. We can observe FINGER achieves significant speed-up over original HNSW graph on all three datasets. Best viewed in color.

Figure 6: Experimental results of ablation studies. Throughput versus Recall@10 chart is plotted for all datasets. HNSW(pecos) is the baseline graph of all other approximating methods. HNSW-FINGER-RP uses random Gaussian basis instead of top eigenvectors of residual vectors for hashing. HNSW-SVD and HNSW-RPLSH directly approximate the distance between vectors whereas HNSW-FINGER approximate the angles of residual vectors. We can observe a significant performance gain of FINGER over all other variants. Best viewed in color.

that on lower recall regions, all methods perform similarly well. All methods could at least speedup vanilla greedy search algorithms 1.25x. But when we look at recalls larger than .8, only FINGER could speedup HNSW over 50% whereas HCNNG or TOGG-KMC failed to accelerate HNSW graph much on SIFT-1M-128 (HCNNG) or GIST-1M-960 (TOGG-KMC). This shows that these previous method might be data sensitive that it only works on certain data distribution. In addition, FINGER remains steadily about 2x speedup on GIST-1M-960 and other two baselines fall to 1.25x quickly. Overall, only FINGER achieves steady and significant speedup ratio over original HNSW method across all datasets. This part of experiments justify that FINGER is a better accelerating search algorithm compared to previous methods.

4.3 Ablation Studies

In this section, we will do two ablation studies to analyze the effectiveness of FINGER. First, we want to demonstrate the proposed basis sampled from top eigenvectors of residual vector matrix indeed performs better than random Gaussian vectors. To justify, we only need to change the way FINGER generates projection basis from learned residual eigenvectors to randomly sampled Gaussian vectors, and we call this method HNSW-FINGER-RP. Comparisons on selected datasets are shown in Figure 6. We can see that on the three selected datasets, HNSW-FINGER all performs 10%-15% better than HNSW-FINGER-RP. This results directly validates the proposed basis generation scheme is better than random Gaussian vectors. Also notice that HNSW-FINGER-RP actually performs better than HNSW(pecos) on all datasets. This further validates that the proposed approximation scheme is useful, and we could use different angle estimations methods to achieve good acceleration.

Second, we want to compare the proposed approximating distance function to other canonical choices. Approximating distance function used in FINGER is based on the decomposition of exact distance and FINGER only estimates the angle of residual vectors. To demonstrate the effectiveness of this approach, we could substitute

the approximating function in line 15 of Algorithm 2 with other approximating distance functions. A natural candidate is directly using RPLSH to estimate the angles of vectors. Another popular candidate is using low-rank SVD to approximate the distance between two vectors. We call these two approaches HNSW-RPLSH and HNSW-SVD. Results are also shown in Figure 6.

As we can see that HNSW-RPLSH and HNSW-SVD performs much worse than FINGER. In fact, these two methods even failed to accelerate HNSW on GLOVE-1.2M-100 dataset. Notice that a major difference between FINGER and these two methods is that HNSW-RPLSH and HNSW-SVD do not use the decomposition introduced in FINGER. HNSW-RPLSH and HNSW-SVD directly approximate the distance between original vectors instead of only the residual vectors part. This will lead to a much larger approximation error and consequently a worse throughput-recall@10 performance. In particular, HNSW-FINGER-RP and HNSW-RPLSH use the same approximation method and the difference is only the target term of approximation. Given the same amount of approximating capability provided by signed locality sensitive hashing, limiting the approximation to only a smaller portion of whole arithmetic would naturally lead to a smaller approximation error. And we can see that the performance difference between HNSW-RPLSH and HNSW-FINGER-RP is significant. This further validates the effectiveness of the approximation scheme proposed in Section 3.2.

5 CONCLUSIONS

In this work, we propose FINGER, a fast inference method for graph-based AKNNS. FINGER approximates distance function in graph-based methods by estimating angles between neighboring residual vectors. FINGER leveraged residual bases to perform memory-efficient hashing estimate of residual angles. The approximated distance can be used to bypass unnecessary distance evaluations, which translates into a faster searching. Empirically, FINGER on top of HNSW is shown to outperform all existing graph-based methods.

6 ACKNOWLEDGEMENTS

REFERENCES

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229
[2] Sunil Arya and David M Mount. 1993. Approximate nearest neighbor queries in fixed dimensions.. In *SODA*, Vol. 93. Citeseer, 271–280.
[3] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.
[4] Franz Aurenhammer. 1991. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)* 23, 3 (1991), 345–405.
[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*. 322–331.
[6] Christopher M Bishop. 2006. Pattern recognition. *Machine learning* 128, 9 (2006).
[7] Deng Cai. 2019. A revisit of hashing algorithms for approximate nearest neighbor search. *IEEE Transactions on Knowledge and Data Engineering* (2019).
[8] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *STOC*. 380–388.
[9] Patrick H Chen, Si Si, Sanjiv Kumar, Yang Li, and Cho-Jui Hsieh. 2019. Learning to screen for fast softmax inference on large vocabulary neural networks. In *ICLR*.
[10] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zhiyong Zheng, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. *NeurIPS* 34 (2021).
[11] DW Dearholt, N Gonzales, and G Kurup. 1988. Monotonic search networks for computer vision databases. In *Twenty-Second Asilomar Conference on Signals, Systems and Computers*, Vol. 2. IEEE, 548–553.
[12] Qin Ding, Hsiang-Fu Yu, and Cho-Jui Hsieh. 2019. A fast sampling algorithm for maximum inner product search. In *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 3004–3012.
[13] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*. 577–586.
[14] Matthijs Douze, Alexandre Sablayrolles, and Hervé Jégou. 2018. Link and code: Fast indexing with graphs and compact regression codes. In *CVPR*. 3646–3654.
[15] Casper Benjamin Freksen. 2021. An Introduction to Johnson-Lindenstrauss Transforms. *arXiv preprint arXiv:2103.00564* (2021).
[16] Cong Fu and Deng Cai. 2016. EFANNA: An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv preprint arXiv:1609.07228* (2016).
[17] Cong Fu, Changxu Wang, and Deng Cai. 2021. High Dimensional Similarity Search with Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.* PP (March 2021).
[18] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. (July 2017). arXiv:cs.LG/1707.00143
[19] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization. *IEEE TPAMI* 36, 4 (2013), 744–755.
[20] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *VLDB*, Vol. 99. 518–529.
[21] Michel X Goemans and David P Williamson. 1995. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)* 42, 6 (1995), 1115–1145.
[22] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *ICML*. PMLR, 3887–3896.
[23] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. 2011. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *Twenty-Second International Joint Conference on Artificial Intelligence*.
[24] Ben Harwood and Tom Drummond. 2016. FANNG: Fast approximate nearest neighbour graphs. In *CVPR*. 5713–5722.
[25] Kaiming He, Fang Wen, and Jian Sun. 2013. K-means hashing: An affinity-preserving quantization method for learning binary compact codes. In *CVPR*.
[26] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*. 604–613.
[27] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. *NeurIPS* 32 (2019).
[28] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE TPAMI* 33, 1 (2010), 117–128.
[29] Zhongming Jin, Debing Zhang, Yao Hu, Shiding Lin, Deng Cai, and Xiaofei He. 2014. Fast and accurate hashing via iterative nearest neighbors expansion. *IEEE transactions on cybernetics* 44, 11 (2014), 2167–2177.
[30] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.
[31] Maximilian Lam. 2018. Word2Bits - Quantized Word Vectors. *arXiv preprint arXiv:1803.05651* (2018).
[32] Der-Tsai Lee and Bruce J Schachter. 1980. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer & Information Sciences* 9, 3 (1980), 219–242.
[33] Conglong Li, Minjia Zhang, David G Andersen, and Yuxiong He. 2020. Improving approximate nearest neighbor search through learned adaptive early termination. In *SIGMOD*. 2539–2554.
[34] Xiaoyun Li and Ping Li. 2019. Random projections with asymmetric quantization. *NeurIPS* 32 (2019).
[35] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE TPAMI* 42, 4 (2018), 824–836.
[36] Etienne Marcheret, Vaibhava Goel, and Peder A Olsen. 2009. Optimal quantization and bit allocation for compressing large discriminative feature space transforms. In *2009 IEEE Workshop on ASRU*. IEEE, 64–69.
[37] Julieta Martinez, Shobhit Zakhmi, Holger H Hoos, and James J Little. 2018. LSQ++: Lower running time and higher recall in multi-codebook quantization. In *ECCV*.
[38] Yusuke Matsui, Yusuke Uchida, Hervé Jégou, and Shin'ichi Satoh. 2018. A survey of product quantization. *ITE Transactions on Media Technology and Applications* 6, 1 (2018), 2–10.
[39] Stanislav Morozov and Artem Babenko. 2019. Unsupervised neural quantization for compressed-domain similarity search. In *ICCV*. 3036–3045.
[40] Javier Vargas Munoz, Marcos A Gonçalves, Zanoni Dias, and Ricardo da S Torres. 2019. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition* 96 (2019), 106970.
[41] Tobias Plötz and Stefan Roth. 2018. Neural nearest neighbors networks. *arXiv preprint arXiv:1810.12575* (2018).
[42] Chanop Silpa-Anan and Richard Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *CVPR*. IEEE, 1–8.
[43] Kohei Sugawara, Hayato Kobayashi, and Masajiro Iwasaki. 2016. On approximately searching for similar word embeddings. In *ACL*.
[44] Hongya Wang, Zhizheng Wang, Wei Wang, Yingyuan Xiao, Zeng Zhao, and Kaixiang Yang. 2020. A Note on Graph-Based Nearest Neighbor Search. *arXiv preprint arXiv:2012.11083* (2020).
[45] Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. 2010. Sequential projection learning for hashing with compact codes. (2010).
[46] Xiang Wu, Ruiqi Guo, Ananda Theertha Suresh, Sanjiv Kumar, Daniel N Holtmann-Rice, David Simcha, and Felix Yu. 2017. Multiscale quantization for fast similarity search. *NeurIPS* 30 (2017), 5745–5755.
[47] Xiaoliang Xu, Mengzhao Wang, Yuxiang Wang, and Dingcheng Ma. 2021. Two-stage routing with optimized guided search and greedy algorithm on proximity graph. *Knowledge-Based Systems* 229 (2021), 107305.
[48] Hsiang-Fu Yu, Cho-Jui Hsieh, Qi Lei, and Inderjit S Dhillon. 2017. A greedy approach for budgeted maximum inner product search. *Advances in neural information processing systems* 30 (2017).
[49] Hsiang-Fu Yu, Kai Zhong, and Inderjit S Dhillon. 2020. PECOS: Prediction for Enormous and Correlated Output Spaces. *arXiv preprint arXiv:2010.05878* (2020).
[50] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)* 52, 1 (2019), 1–38.
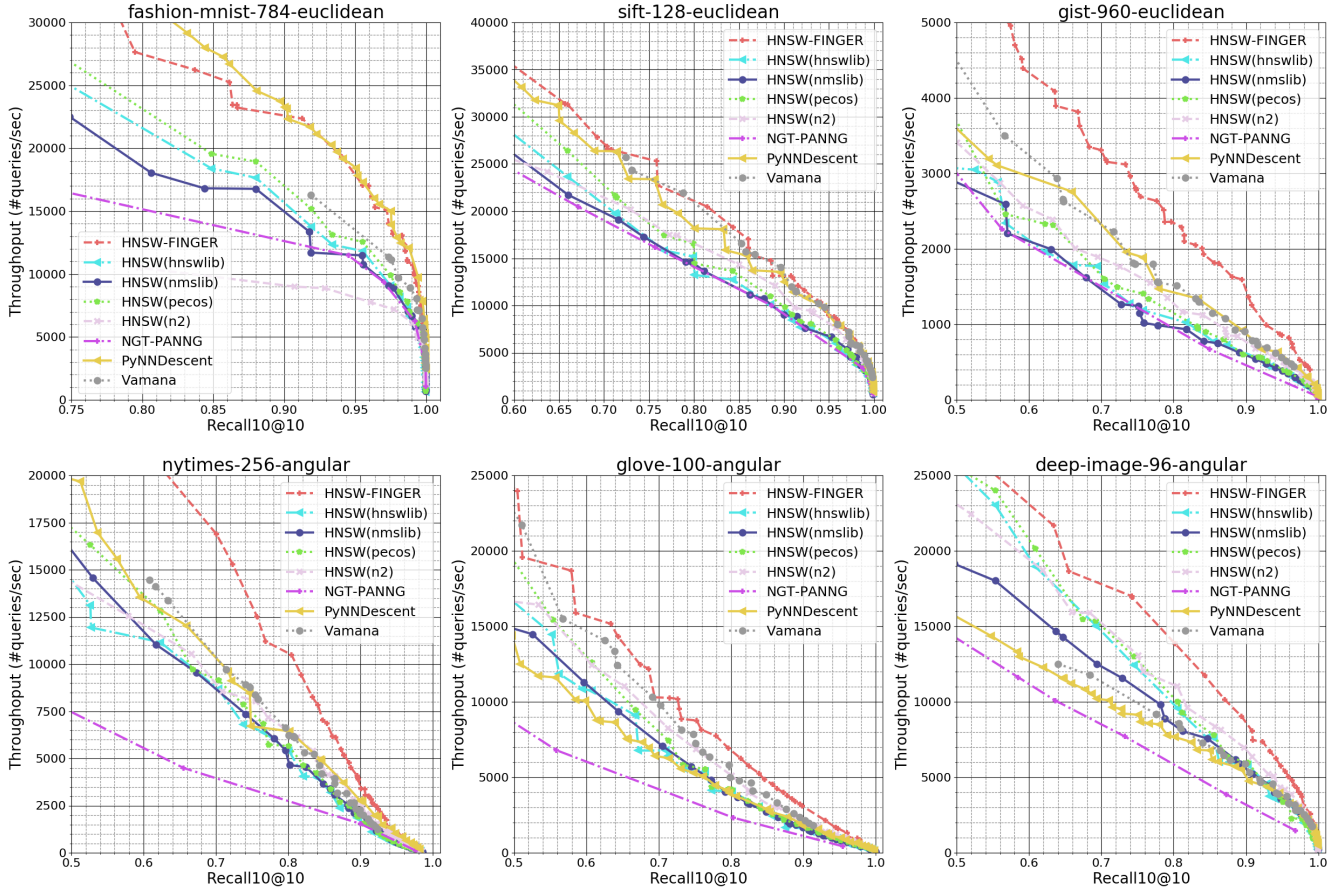
Figure 7: Experimental results of all graph-based methods. Throughput versus Recall@10 chart is plotted for all datasets. Top row presents datasets with $L2$ distance measure and bottom row presents datasets with angular distance measure. We can observe a significant performance gain of HNSW-FINGER over existing graph-based methods. Best viewed in color.

A COMPLETE COMPARISON OF GRAPH-BASED METHODS

Complete results of all graph-based methods are shown in Figure 7. HNSW-FINGER basically outperforms all existing graph-based methods except on FashionMNIST-60K-784 where PyNNDescent performs extremely well. Results show that currently no graph-based methods completely exploits the training data distribution. This reflects the importance of the inference acceleration methods as FINGER that can create consistently faster inference on all underlying search graph. Making a search graph maximally suitable for applying FINGER is also an interesting future direction. In principle, FINGER could also be applied on PyNNDescent to further improve the result. For example, applying FINGER code on kNN graphs from PyNNDescent. At Recall@10 of 99%, on the Fashion-MNIST dataset, FINGER improved the throughput by 40% over the original PyNNDescent and HNSW. For the SIFT dataset, FINGER improved the throughput by 20% and 25% over the original PyNNDescent and HNSW, respectively.

B DETAILED STEPS OF FINGER APPROXIMATION

As shown in Eq.(2), the L2 distance between q and d can be written as:

$$\|q - d\|_2^2 = \|(q_{proj} - d_{proj})\|_2^2 + \|q_{res}\|_2^2 + \|d_{res}\|_2^2 - 2q_{res}^T d_{res},$$

where $q_{proj}, d_{proj}, q_{res}, q_{res}$ are obtained by projecting onto vector of center node. We will explain each term individually and use **bold text** to denote information could be pre-computed and stored. We will also analyze number of arithmetic and memory read operations needed for the whole algorithm.

- $\|(q_{proj} - d_{proj})\|_2^2$: Since q_{proj} and d_{proj} are projections of q and d onto the vector of the center node. Without loss of generality, we can write it as $q_{proj} = tc$ and $d_{proj} = bc$, where c is the center vector and t, b are scalars. $\|(q_{proj} - d_{proj})\|_2^2$ then becomes $(t - b)^2 \|c\|_2^2$. **We can pre-compute $\|c\|_2^2$ for each node** and $(t - b)^2$ is just a subtraction plus a multiplication to itself. b **for each neighboring node can also be pre-calculated**. To get t, recall the projection formula: $t = \frac{q^T c}{\|c\|_2^2}$. **The denominator $\|c\|_2^2$**

is pre-computed so we only need to get the result of the inner-product between q and c. In Section 3.2, we explained when we explore neighbors of center node, we must have visited it before so **the value $\|q - c\|_2^2$ is stored in candidate queue**; therefore, we can get $q^T c = \frac{\|q\|^2 + \|c\|^2 - \|q-c\|^2}{2}$, and thus $t = \frac{q^T c}{\|c\|_2^2}$ with simple calculations. Notice that $\|q\|_2^2$ is a one time cost for all nodes the cost is negligible when dataset is moderately large. In total, we need 3 memory reads and 6 arithmetic to complete this step.

- $\|dres\|_2^2$: $\|dres\|_2^2$ **can be pre-computed** and stored as a single floating point so no computation is needed here. It costs 1 memory read.
- $\|q_{res}\|_2^2$: From above, we know that $\|q_{proj}\|_2^2 = t^2 \|c\|_2^2$, and **we have already loaded the pre-computed** $\|c\|_2^2$. Thus to get $\|q_{proj}\|_2^2$, we need 2 multiplications. Since $\|q\|_2^2 = \|q_{proj}\|_2^2 + \|q_{res}\|_2^2$, we can get $\|q_{res}\|_2^2 = \|q_{proj}\|_2^2 - \|q\|_2^2$, by an additional subtraction. In total, it costs 3 arithmetic.
- $q_{res}^T d_{res}$: we get this term by using $\|q_{res}\|_2 \|d_{res}\|_2 \cos(q_{res}, d_{res})$. Given the LSH basis B, $\mathbf{sgn}(d_{res}^T B)$ **can be pre-computed as saved as compact binary representations.** To get $q_{res}^T B$, recall $q_{res} = q - q_{proj}$, so $q_{res}^T B = q^T B - q_{proj}^T B = q^T B - tc^T B = q^T B - \frac{q^T c}{\|c\|_2^2} c^T B$. **results of $c^T B$ can be pre-computed, and we have already calculated** $\frac{q^T c}{\|c\|_2^2}$, so we can get $q_{res}^T B$ by r subtractions of $q_{proj}^T B$ from $q^T B$, where r is the number of LSH basis used. Again, computing $q^T B$ is also a one time cost for all nodes in the search of a query, so the cost is negligible when dataset is moderately large. After getting $q_{res}^T B$, we can take its sign and we are ready to estimate angles. Notice that this whole process needs to be done only once for a center node exploration. When number of edges is moderately large (i.e., 32 or 64), this cost is almost negligible for each neighboring node. Without loss of generality, we assume calculating hamming distance between $\text{sgn}(d_{res}^T B)$ and $\text{sgn}(q_{res}^T B)$ costs r arithmetic. **We can pre-compute $\|d_{res}\|_2$ and q_{res} has been calculated above**. So we just need 2 more multiplications to get $q_{res}^T d_{res}$. In total, we need $r + \frac{r}{32} + 1$ memory reads, $r + 2$ arithmetic to complete this step.

Since $\|q - d\|_2^2 = \|(q_{proj} - d_{proj})\|_2^2 + \|q_{res}\|_2^2 + \|d_{res}\|_2^2 - 2q_{res}^T d_{res}$, we need 4 more arithmetic to combine all above terms. Thus in total it costs $r + \frac{r}{8} + 5$ memory reads and $r + 15$ arithmetic to complete the computation. Consider a full dimensional L2 distance on SIFT-1M-128 dataset. Recall the data dimension $m = 128$ and we use $r = 64$. L2 distance requires 128 memory reads, 128 subtractions, 128 multiplications and 127 additions. For approximation distance, in total we only need 71 memory reads and 80 arithmetic. We can observe that approximation distance used much less operations so it will be much faster.

Time Complexity Analysis. In theory, the time complexity of graph search is linear to the number of visited nodes times the distance computation cost between query and each node. The former is query dependent (non analytic), while the latter is where the improvement is made in this paper. Specifically, the time complexity of distance computation is reduced from O(d) to O(r), where r is the low rank used in LSH (Lemma 1) and d is the original data dimension.

C MEMORY FOOTPRINT AND OVERHEAD OF HNSW-FINGER AND HNSW

As illustrated in Section B, we pre-compute and store r floating points in $c^T B$ ($r \times 4$ bytes) and $\|c\|_2^2$ (1 byte) for each node. For each edge, we pre-compute and store $\frac{r}{8}$ bytes of signed code and projection coefficient b (4 bytes) and residual norm d_{res} (4 bytes). Thus in total for a graph $G = (V, E)$, we save additional $|V| \times (4r + 1) + |E| \times (\frac{r}{8} + 8)$ bytes. Take GIST-1M-960 with maximal 96 edges per node for example, we use $r = 64$, $|E|$ is maximally $1 \times 96 = 96$ million edges. This translates into about additional $1e^6 \times (4 * 64 + 1) + 96e^6 (64/8 + 8)$ Bytes (≈ 1709 MB), which is about the half whole original 1M training database (3.6GB). Compared to the original HNSW model (4.5GB), the additional cost of FINGER is acceptable. In particular, if we use full precision low-rank model, even for $r = 16$, it will cost $|E| \times 16 \times 4$ Bytes (≈ 5859 MB). We can use much more basis in RPLSH setup with less memory footprint. Notice that this setup is perhaps the largest working search index for all the 6 datasets used in this paper. Best performing graph mostly will not need to have more than 96 edges and 64 basis. Thus, in practice, additional storage is about a constant of original training vector size. In terms of pre-processing time, with the same hardware configuration, on the GIST dataset, the time overhead of FINGER and TOGG-KMC over HNSW index building are 10.08% and 11.05%, respectively, which is not very time consuming compared to previous baseline methods.

D FORMULATION OF INNER-PRODUCT

In the main text, we presented derivation of $L2$ distance, and in this section we will derive the approximation for inner-product distance measure. Notice that angle measure can be obtained by firstly normalizing data vectors and then apply inner-product distance and thus the derivation is the same. For a query q and data point d, inner-product distance measure is $Dist = q^T d$. Similar to $L2$ distance, we can apply the same decomposition to write $q = q_{proj} + q_{res}$ and $d = d_{proj} + d_{res}$. substituting the decomposition into distance definition, we have

$$Dist = q_{proj}^T d_{proj} + q_{res}^T d_{res}.$$

As in $L2$ case q_{proj} and d_{proj} can be obtained by simple operations and the remaining uncertainy term is again $q_{res}^T d_{res}$. Therefore, in inner-product case, angle between neighboring residual vectors is still the target to approximate.