Revisiting the Index Construction of Proximity Graph-Based Approximate Nearest Neighbor Search

Shuo Yang[§]
Xidian University
yangsh@stu.xidian.edu.cn

Jiadong Xie[§]
The Chinese University of Hong Kong
jdxie@se.cuhk.edu.hk

Yingfan Liu[*]
Xidian University
liuyingfan@xidian.edu.cn

Jeffrey Xu Yu
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk

Xiyue Gao
Xidian University
xygao@xidian.edu.cn

Qianru Wang
Xidian University
wangqianru@xidian.edu.cn

Yanguo Peng
Xidian University
ygpeng@xidian.edu.cn

Jiangtao Cui
Xidian University
cuijt@xidian.edu.cn

ABSTRACT

Proximity graphs (PG) have gained increasing popularity as the state-of-the-art solutions to k-approximate nearest neighbor (k-ANN) search on high-dimensional data, which serves as a fundamental function in various fields, e.g., retrieval-augmented generation. Although PG-based approaches have the best k-ANN search performance, their index construction cost is superlinear to the number of points. Such superlinear cost substantially limits their scalability in the era of big data. Hence, the goal of this paper is to accelerate the construction of PG-based methods without compromising their k-ANN search performance.

To achieve this goal, two mainstream categories of PG are revisited: relative neighborhood graph (RNG) and navigable small world graph (NSWG). By revisiting their construction process, we find the issues of construction efficiency. To address these issues, we propose a new construction framework with a novel pruning strategy for edge selection, which accelerates RNG construction while keeping its k-ANN search performance. Then, we integrate this framework into NSWG construction to enhance both the construction efficiency and k-ANN search performance of NSWG. Extensive experiments are conducted to validate our construction framework for both RNG and NSWG, and that it significantly reduces the PG construction cost, achieving up to 5.6x speedup, while not compromising the k-ANN search performance.

PVLDB Reference Format:
Shuo Yang, Jiadong Xie, Yingfan Liu, Jeffrey Xu Yu, Xiyue Gao, Qianru Wang, Yanguo Peng, and Jiangtao Cui. Revisiting the Index Construction of Proximity Graph-Based Approximate Nearest Neighbor Search. PVLDB, 18(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/xdyangsh/FastKCNA.

[§] Shuo Yang and Jiadong Xie are the joint first authors.
[*] Yingfan Liu is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Recent breakthroughs in deep learning models, specifically embedding models, have revolutionized the representation of various types of data, such as images and text chunks. These models transform the data into vectors, which encapsulate key information in a high-dimensional space for semantic analysis. As a result, k-approximate nearest neighbor (k-ANN) search on high-dimensional vectors has emerged as a fundamental problem across multiple domains, including information retrieval [20, 24], recommendation system [44] and large language models [4, 8, 22, 31, 34]. In particular, due to the popularity of retrieval-augmented generation (RAG) [4, 31], k-ANN search on high-dimensional vectors causes more and more attention as a key RAG component. Given a dataset $D \subset \mathbb{R}^d$ and a query vector $q \in \mathbb{R}^d$, k-ANN search returns the k vectors in D that are sufficiently close to q, where k is a specified parameter.

k-ANN search has been extensively studied for several decades, hence there exists a wealth of methods in the literature [12, 18, 33, 36, 38, 52, 56, 58]. According to recent studies [5, 32, 33, 58], the proximity graph (PG) based methods [12, 15–17, 33, 38, 39, 48] have emerged as the promising solution and have demonstrated superior performance compared to other approaches, such as hashing-based methods [18, 25, 36, 37, 51, 52], invert index-based methods [7, 29] and tree-based methods [9, 10, 30, 42]. A PG treats each vector $u \in D$ as a graph vertex and then connects edges between u and its close neighbors. Notably, all PGs share the same vertex set but have various edge sets due to their distinct edge-selection strategies. According to previous studies [33, 47, 48, 58], PGs are divided into three categories by different neighbor-selection strategies: k nearest neighbor graph (KNNG), relative neighborhood graph (RNG), and navigable small world graph (NSWG).

Although PG-based approaches have superior search performance, they still suffer from a significantly higher cost in terms of index construction than other methods. This is primarily due to the necessity of identifying close neighbors for each point to establish the graph edges. In the traditional applications of k-ANN search, PG index is built offline and then responds to k-ANN queries. Thus its construction cost is treated as a second-class performance indicator. However, in the emerging scenario, i.e., the RAG model training [4, 22, 26], the PG index will be frequently built in an online manner, due to the tuning on the embedding model that transforms the source data such as text chunks into vectors. As a result, it is urgent to build PG efficiently while maintaining search

Table 1: PG index comparisons on Gist1M

Approaches	Building Time (s)	QPS when Recall@10=			
		0.85	0.90	0.95	0.99
NSG [17]	573	**971**	**712**	**417**	133
LSH-APG [62]	433	453	305	172	53
DiskANN [50]	214	657	469	266	83
ParlayANN [40]	409	669	472	274	89
RNN-Descent [45]	**121**	739	539	345	**139**
FastNSG (ours)	119	964	694	402	148

performance. To address this issue, several studies have been proposed to accelerate the index construction, such as DiskANN [50], RNN-Descent [45], LSH-APG [62] and ParlayANN [40]. However, as shown in Table 1, their search performance remains noticeably inferior to NSG [17], where NSG is one of the state-of-the-art (SOTA) methods that do not prioritize index construction. In Table 1 on Gist1M, the second column shows the time of index construction and the next four columns depict queries per second (QPS) at different recall levels (higher is better). Hence, they sacrifice the search performance to expedite the index construction, which violates the primary goal of k-ANN search.

In this work, we begin by revisiting the construction process of SOTA PGs, namely RNG and NSWG. On top of the KNNG, RNG is built with three key phases. In the first phase, an initial KNNG G_{k_0} is constructed (initialization phase), and then the results of k-ANN search on G_{k_0} for each node are obtained as candidate neighbors (search phase), which are further refined to prune redundant edges and enhance connectivity (refinement phase). Unlike RNG assuming the priori knowledge of the whole dataset, NSWG builds from scratch and inserts nodes incrementally into the current graph one by one. Like RNG, NSWG shares both search phase that finds close neighbors in the current incomplete graph index and refinement phase that builds edges between each node and a selected subset of those neighbors found.

Up on the revisiting, we identify the PG construction issues and further propose an efficient construction framework for RNG, which could also be used to accelerate the NSWG index construction. To be specific, our contributions are summarized as follows. ❶ During revisiting the index construction procedures of current PGs, we analyze the essential operation in PG construction, i.e., size-k candidate neighbor set acquisition (k-CNA), whose quality is key to the finally built PG. RNG obtains k-CNA results by initialization phase and search phase, while NSWG by search phase. However, we find issues related to k-CNA, i.e., (1) the inefficiency of obtaining k-CNA results in RNG, and (2) the poor k-CNA quality in NSWG. ❷ To tackle the inefficiencies of k-CNA in RNG, we introduce a new RNG construction framework featuring a novel pruning strategy aimed at improving the efficiency of k-CNA while preserving its quality. ❸ To enhance the quality of k-CNA in NSWG, primarily caused by its node-by-node insertion method, we substitute this method with a layer-by-layer insertion strategy, and integrate it with our RNG construction framework to further enhance its efficiency. ❹ We conduct extensive experiments on real-life datasets to validate the effectiveness of our construction framework. The results demonstrate that our framework accelerates the construction of the representative RNG index NSG and the representative NSWG index HNSW up to 5.6x and 4.6x respectively, while achieving comparable or even better search performance.

Table 2: Summary of notations and abbreviations

	Definition
D	the set of n d-dimensional vectors
$G = (V, E)$	a proximity graph with vertex set V and edge set E
$N_G(u)$	the set of out-neighbors of u in G
$V(G)/E(G)$	the node/edge set of G
q	a query data point
k	the number of returned results in k-ANN search
L	the pool width in k-ANN search of PG
ep	the entry point in k-ANN search of PG
$C(u)$	the k-CNA results of u
M	the upper bound of node out-degrees in PG
ef	the key construction parameter of HNSW
α	the angle threshold in α-pruning
NSG	Navigating Spread-out Graph
HNSW	Hierarchical Navigable Small World

The paper is organized as follows. We provide the preliminaries in Section 2. In Section 3, we revisit the existing PG methods and identify their construction issues. In Section 4, we propose the refinement-before-search scheme as the basis of our construction framework for RNG and NSWG as shown in Section 5. We present our experimental studies in Section 6. Furthermore, we discuss the related works in Section 7 and conclude our work in Section 8.

2 k-ANN SEARCH AND PROXIMITY GRAPHS

k-ANN Search: Let $D \subset \mathbb{R}^d$ be a high-dimensional dataset consisting of n d-dimensional points. We denote the L2 norm (i.e., Euclidean distance) between two points $u, v \in \mathbb{R}^d$ as $dist(u, v)$. Given a dataset D and a query point $q \in \mathbb{R}^d$, *the k-approximate nearest neighbor (k-ANN)* search aims to find the top-k points in D with the minimum distances from the query q. This paper focuses on the in-memory solutions, which assume that D and the corresponding index can be hosted in the memory [16, 17, 33, 38, 39, 48]. Table 2 summarizes the notations.

According to recent studies [5, 32, 33, 58], proximity graph (PG) based approaches are the SOTA methods for k-ANN search. In the following, we discuss PG-based approaches and their search algorithm for answering k-ANN queries.

2.1 Proximity Graph

A PG $G = (V, E)$ of D is a directed graph, where each node in V uniquely represents a vector (i.e., data point) in D, and two nodes are connected by an edge in E if their corresponding vectors are close to each other. For a node $u \in V$, we use $N_G(u)$ to denote the set of out-neighbors of node u in a PG G.

As presented in previous studies [33, 47, 48, 58], PGs are classified into three categories according to their edge-selection strategies.

k-Nearest Neighbor Graph (KNNG): Each node in KNNG is connected to its k-approximate nearest data points, which is proposed as an approximation of the Delaunay graph (DG) [6], since DG becomes a complete graph when the dimension is large [23, 48].

Relative Neighborhood Graph (RNG): RNG is constructed based on KNNG, it removes the longest edge in every possible triangle in the KNNG, i.e., if edge (u, v) exists in RNG, there exists

Algorithm 1: KANNSearch(G, q, k, L, ep)

Input : graph index G, query point q, k for top-k, pool width L and entering point ep
Output : k-ANN of query point q

1 $i \leftarrow 0$;
2 $pool[0] \leftarrow (ep, dist(q, ep))$;
3 **while** $i < L$ **do**
4 $u \leftarrow pool[i]$;
5 **for** each $v \in N_G(u)$ **do**
6 insert $(v, dist(q, v))$ into $pool$;
7 sort $pool$ and keep the L closest neighbors;
8 $i \leftarrow$ index of the first unexpanded vertex in $pool$;
9 **return** $pool[0, \ldots, k-1]$

no edge (u, w) in the graph such that $dist(u, w) < dist(u, v)$ and $dist(v, w) < dist(u, v)$. RNG enhances its connectivity by adding extra edges between connected components, while limiting the out-degree of each node to a small constant [17].

Navigable Small World Graph (NSWG): NSWG is derived from Milgram's social experiment [53], which demonstrates that two nodes in a large graph are connected by a short path that can be discovered through greedy routing. Hence, the NSWG construction involves incrementally node-by-node insertion, where each node is connected to its k-ANN in the current incomplete graph.

2.2 Beam Search

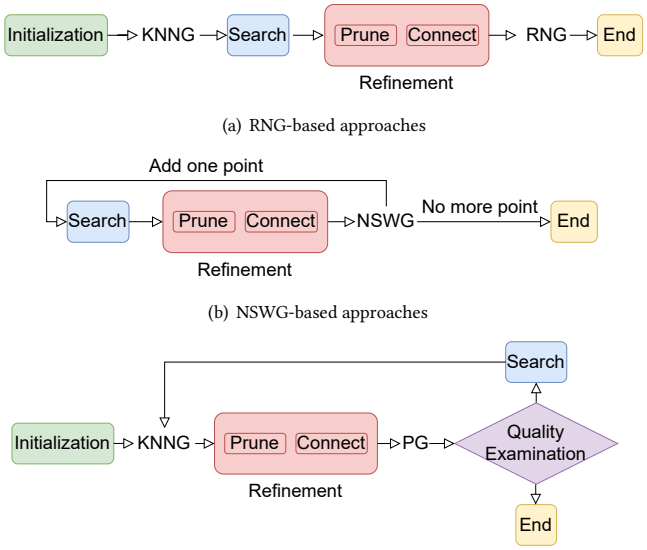
Although PGs have different construction algorithms, they share the same search method called beam search, which is on top of greedy routing. Greedy routing involves selecting the nearest out-neighbor of the current node at each step and terminates when no closer out-neighbors are available.

To extend the greedy routing for k-ANN search, beam search (i.e., the best-first search) is thus proposed. Specifically, as shown in Algorithm 1, the search process starts from an entering point ep and puts it in a sorted array $pool$ of nodes, which is maintained to store the currently found L-closest neighbors (Lines 1-2). Then, it iteratively extracts the closest but unexpanded neighbor u from $pool$ (Line 4) and expands u to refine $pool$, until the termination condition is satisfied (Line 3). In each iteration, expanding u for q is shown in Lines 5-7, where each neighbor $v \in N_G(u)$ is treated as a k-ANN candidate of q (Line 5) and further verified by an expensive distance computation (Line 6) to refine $pool$ (Line 7). At the end of each iteration (Line 8), the algorithm finds the closest but unexpanded vertex in $pool$ as the next one to be expanded. It terminates when the first L vertices in $pool$ have been expanded (Line 3).

We can see that L plays a crucial role in the trade-off between efficiency and accuracy, i.e., a higher L leads to more accurate results but a larger time cost. Furthermore, beam search is a key component applied to answer k-ANN search in the PG construction, as discussed in the next section.

3 PG CONSTRUCTION REVISITED

In this section, we revisit the index construction of the SOTA PG approaches, namely RNG methods [16, 17, 48, 50] and NSWG methods [38, 39, 62]. We first present the construction process of RNG and NSWG respectively, followed by identifying their issues in building efficiency.



(a) RNG-based approaches

(b) NSWG-based approaches

(c) Our framework

Figure 1: The construction phases

3.1 RNG Index Construction

To construct an RNG index, the computation of distances between each pair of nodes is necessary in order to eliminate the longest edge in all possible triangles. Unfortunately, its time complexity is $O(n^2 d)$, which is practically impossible for large data. Hence, some studies propose to construct a practical version of RNG via another PG, such as KNNG [16, 17], NSG [48] or even a random graph [50].

In this section, we take NSG as an RNG representative. This is because NSG is widely acknowledged as one of the SOTA methods for k-ANN search [48, 58], and recent studies focus on improving search performance by making minor modifications on NSG, e.g., extending the pruning strategy [16, 48].

In a nutshell, as shown in Figure 1(a), the construction of NSG involves three phases: initialization, search and refinement. The initialization phase focuses on constructing an approximate KNNG, while search phase aims to improve the quality of the KNNG, i.e., enhancing the accuracy of neighbors (k-ANN) for each node in the graph. Finally, the refinement phase incorporates prune and connect operations to reduce the node out-degree and enhance the graph connectivity respectively. We present the details of NSG construction process in Algorithm 2.

Initialization Phase: An initial KNNG G_{k_0} where each node has k_0 neighbors is built in initialization phase by the SOTA method called KGraph [15], as shown in Line 1 of Algorithm 2. The purpose of this phase is to build an index for search phase.

Search Phase: As in Lines 3-4 of Algorithm 1, for each $u \in D$, the search phase performs k-ANN search on G_{k_0} in order to obtain the candidate set $C(u)$ for refinement. Notably, each k-ANN search starts from the entering point ep which is the closest point in D to the centroid of D (Line 2).

Algorithm 2: BuildNSG(D, k_0, k, L, M)

Input : dataset D and four parameters k_0, k, L and M
Output : an NSG G

/* **Phase 1:** Initialization */
1 build G_{k_0} with k_0 neighbors by KGraph [15];
/* **Phase 2:** Search */
2 $ep \leftarrow$ KANNSearch (G_{k_0}, cn, k, L, rn), where cn is the centroid of D and $rn \in D$ is a random node;
3 **for** each $u \in D$ in parallel **do**
4 $\quad C(u) \leftarrow$ KANNSearch (G_{k_0}, u, k, L, ep);
/* **Phase 3:** Refinement (G =Refine $(\{C(u)|u \in D\}, M)$) */
5 **for** each $u \in D$ in parallel **do**
6 $\quad N_G(u) \leftarrow$ Prune $(u, C(u), M)$;
7 **for** each $u \in D$ in parallel **do**
8 $\quad N_G(u) \leftarrow$ Prune $(u, N_G(u) \cup \{v|u \in N_G(v)\}, M)$;
9 find connected components via DFS;
10 add extra edges into $E(G)$ between connected components;
11 **return** G

Algorithm 3: Prune($u, C(u), M$)

Input : a vertex u, neighbor set $C(u)$ and out-degree limit M
Output : a pruned neighbor set of u

1 $PrunedNeighbor \leftarrow \emptyset$;
2 **for** each $v \in C(u)$ in the ascending order of $dist(u, v)$ **do**
3 $\quad DominateFlag \leftarrow$ false;
4 $\quad$ **for** each $w \in PrunedNeighbor$ **do**
5 $\quad\quad$ **if** $dist(v, w) < dist(u, v)$ **then**
6 $\quad\quad\quad DominateFlag \leftarrow$ true;
7 $\quad$ **if** $DominateFlag$ = false **then**
8 $\quad\quad PrunedNeighbor \leftarrow PrunedNeighbor \cup \{v\}$
9 $\quad$ **if** $|PrunedNeighbor| \geq M$ **then break**;
10 **return** $PrunedNeighbor$

Refinement Phase: This phase removes redundant neighbors in the set $C(u)$ via a pruning strategy (Algorithm 3) to obtain $N_G(u)$ with the constraint $|N_G(u)| \leq M$, where M is a specific threshold (Lines 5-6). To improve the graph connectivity, unidirectional edges are added between u and each $v \in N_G(u)$, which might trigger an extra pruning process in order to limit the out-degree of u (Lines 7-8). Finally, a depth-first search (DFS) is employed to identify any remaining connected components in G, and additional edges are then added to connect them together (Lines 10-11).

The widely used pruning process focuses on eliminating the longest edge within each possible triangle formed by the points in the dataset. For simplicity, we call this strategy as *RNG pruning*. Specifically, if edge (u, v) exists in the NSG only if v is not dominated by any neighbor w of u, i.e., there is no edge (u, w) such that $dist(u, w) < dist(u, v)$ and $dist(v, w) < dist(u, v)$. In the practical version of NSG, this pruning process for each node u is modified in two aspects, i.e., (1) the out-neighbors of each u are only picked from the close neighbor set $C(u)$ and (2) each node has at most M out-neighbors. The first modification improves the construction efficiency, while the second accelerates k-ANN search in NSG. The details of are presented in Algorithm 3. Each candidate neighbor

Algorithm 4: BuildHNSW(D, ef, M)

Input : dataset D and two parameters ef and M
Output : an HNSW G

1 $m_L \leftarrow 0$;
2 initialize G_0 with a randomly selected point $v \in D$ and no edges;
3 **for** each $u \in D \setminus \{v\}$ in parallel **do**
4 $\quad$ randomly determine the highest layer of u is l;
5 $\quad$ **if** $l > m_L$ **then**
6 $\quad\quad m_L \leftarrow l$; $ep \leftarrow u$;
$\quad$ /* **Phase 1:** Search */
7 $\quad w \leftarrow ep$;
8 $\quad$ **for** each $i \leftarrow m_L$ downto $l + 1$ **do**
9 $\quad\quad w \leftarrow$ KANNSearch$(G_i, u, 1, 1, w)$;
10 $\quad W_{l+1} \leftarrow \{w\}$;
11 $\quad$ **for** $i \leftarrow l$ downto 0 **do**
12 $\quad\quad W_i \leftarrow$ KANNSearch$(G_i, u, ef, ef, W_{i+1}[0])$;
$\quad$ /* **Phase 2:** Refinement */
13 $\quad$ **for** $i \leftarrow l$ downto 0 **do**
14 $\quad\quad$ **for** each $u \in V(G_i)$ **do**
15 $\quad\quad\quad N_{G_i}(u) \leftarrow$Prune (u, W_i, M);
16 $\quad\quad$ **for** each $u \in V(G_i)$ **do**
17 $\quad\quad\quad N_{G_i}(u) \leftarrow$Prune $(u, N_{G_i}(u) \cup \{v|u \in N_{G_i}(v)\}, M)$;
18 **return** $G = \{G_0, G_1, \ldots, G_{m_L}\}$

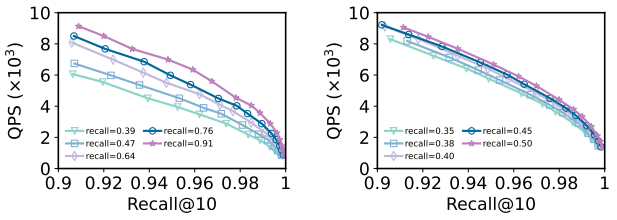
$v \in C(u)$ is checked individually, in ascending order of $dist(u, v)$ (Line 2). v is selected as a neighbor of u only if it is not dominated by any existing neighbors (Lines 3-8), and the process terminates once M neighbors have been selected (Line 9).

As to other RNG methods such as DPG [33] and τ-MNG [48], we can build them by only replacing the RNG pruning strategy (Algorithm 3) with their own ones. Hence, the mainstream RNG methods follow the same framework of index construction.

3.2 NSWG Index Construction

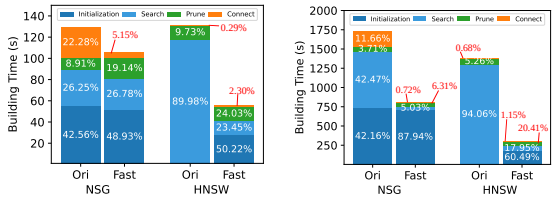
Unlike RNG, NSWG is constructed by incrementally inserting nodes into the current graph and connecting each node to a subset of its k-ANN found in the current graph. Following this idea, several methods such as NSW [38], HNSW [39] and LSH-APG [62] are proposed, where HNSW [39] stands out as the SOTA approach. Hence, in this section, we focus on the HNSW construction process.

As illustrated in Figure 1(b), the construction of HNSW involves two phases for each inserted point: search and refinement. The details are presented in Algorithm 4. HNSW begins by initializing the graph with a single point (Lines 1-2). For each remaining point, HNSW randomly determines its highest layer l using an exponentially decaying probability distribution (Line 4). Like NSG, the search phase focuses on finding a candidate neighbor set for each node u. It starts the search from the top layer down to layer $l + 1$ via greedy routing (Lines 8-9) and performs k-ANN search on each lower layer to obtain the candidate neighbor set for refinement (Lines 10-12). Next, for each lower layer (from l to 0), HNSW applies the RNG pruning strategy (Algorithm 3) to prune the neighbors obtained from the search phase. Like NSG, HNSW adds undirectional edges between the inserted node and its selected neighbors



(a) NSG on Sift1M (b) HNSW on Sift1M

Figure 2: The effects of the k-CNA quality (average recall of every node) on the search performance of the derived PG



(a) Sift1M (b) Gist1M

Figure 3: The cost decomposition of PG construction

to enhance connectivity, while limiting the out-degree of each node to a specific number M (Lines 16-17). However, there is no connect operation in the refinement phase of HNSW.

3.3 PG Construction Issues

As discussed above, we can see that both RNG and NSWG construction share the same procedure for each node $u \in D$, i.e., finding a set of k close neighbors of u, denoted *size-k candidate neighbor set acquisition* (k-CNA), and then derive the PG by the refinement phase that takes the k-CNA results as input. Specifically, the RNG construction method combines initialization phase and search phase to generate the k-CNA results, while the search phase of NSWG construction performs k-ANN search on the current incomplete graph index for k-CNA results. Hence, *both RNG and NSWG are derived from the k-CNA results in* refinement *phase.*

Due to the importance of the k-CNA results, there are two performance aspects of obtaining k-CNA, i.e., efficiency and quality. First, the cost of obtaining k-CNA results contributes to the total construction cost and thus its efficiency is key to the construction efficiency. Second, the k-CNA quality, measured by the average over the recall of the k-CNA neighbors for each node $u \in D$ w.r.t. its exact k nearest neighbors, significantly affects the k-ANN search performance of the graph index derived from k-CNA results. We present such an effect through the following experimental study.

The Importance of k-CNA Quality: For the PG derived from k-CNA results by refinement phase, there are two aspects of its search performance, i.e., efficiency by queries per second (QPS) and accuracy by *Recall*@10 of returned k-ANN. As depicted in Figure 2, each curve represents the search performance of a derived PG with a distinct recall of k-CNA results. The results clearly indicate that the k-CNA quality significantly impacts the search performance of the derived PG. Specifically, the recall of k-CNA results exhibits a positive effect on the search performance of PG. Such an effect could also observed on other PGs such as DPG and τ-MNG. Notably, the k-CNA quality of HNSW is limited to 0.5 due to its incremental node-by-node insertion strategy of index construction.

Considering the strong relationship between the quality of k-CNA and search performance, we identify two issues in the construction of RNG and NSWG respectively.

RNG Construction Issue: *RNG methods suffer from inefficiency in obtaining k-CNA results*, caused by search phase that takes the initial KNNG generated by initialization phase as the graph index for accuracy improvement. As demonstrated in previous experimental studies [33, 58], with similar (e.g. tens of) average out-degrees, KNNG is more prone to local optima than RNG and NSWG due to directional edges and weak connectivity. To address this, RNG equips the initial KNNG with a pretty large (e.g. hundreds of) out-degree, which enhances the k-CNA quality but leads to an inefficiency issue. As depicted in Figure 3, the search on KNNG to enhance the k-CNA quality constitutes a significant portion of the overall cost of building the representative RNG method (i.e., NSG), which even surpasses that taken for initialization on the Gist1M dataset. A similar phenomenon could be found in other RNG methods such as τ-MNG.

NSWG Construction Issue: *NSWG methods suffer from a poor k-CNA quality in construction*, caused by its building strategy of incremental node-by-node insertions: utilizes the current graph index with only a part of nodes to conduct k-ANN search for k-CNA results. Hence, the expected value of k-CNA quality in NSWG is only 0.5 even if all the k-ANN queries are answered correctly. Since achieving exact correctness in k-ANN queries is not feasible, the average recall in practice is upper bounded by 0.5, as demonstrated in Figure 2(b).

4 REFINEMENT BEFORE SEARCH

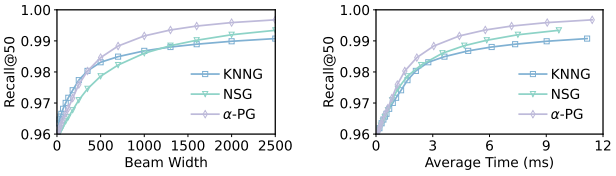
In this section, we focus on addressing the issue identified in the last section regarding the RNG construction. At a high level, we propose replacing the *search-before-refinement* scheme (Figure 1(a)) with a *refinement-before-search* scheme (Figure 1(c)) in RNG construction. To enhance the efficiency of acquiring high-quality k-CNA results, we introduce a novel pruning strategy, α-pruning for neighbor selection in refinement (Section 4.1). Then, we theoretically analyze our proposed scheme to demonstrate its efficacy (Section 4.2).

4.1 α-Pruning Strategy

To enhance the k-CNA efficiency in RNG, we do not use the initial KNNG G as the index used in search phase due to its large out-degree. In our new scheme, we conduct the first refinement on KNNG G to obtain an RNG index denoted as $\hat{G}$ with a much smaller out-degree, then conduct the search phase on $\hat{G}$ to obtain the k-CNA results, which are further used to produce the final RNG index via the second refinement.

However, altering the order of refinement and search directly is unsuitable for index construction due to the following issue.

RNG Pruning Issue: In Section 3, we discuss the RNG pruning strategy used in the refinement as shown in Figure 5, the edge (u, v) will be pruned if there exists edge (u, w) such that $dist(u, w) <$



(a) Recall v.s. Beam Width (b) Recall v.s. Time

Figure 4: Comparing k-CNA results on Glove.

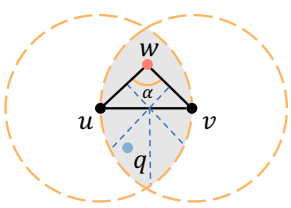


Figure 5: An example of α-pruning strategy

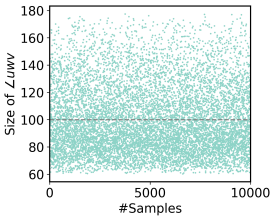


Figure 6: The size of $\angle uwv$ by Monte Carlo simulations

$dist(u,v)$ and $dist(v,w) < dist(u,v)$. However, as shown in the following example, it is primarily not designed for k-CNA.

Example 4.1: As shown in Figure 5, w and v are out-neighbors of u in KNNG, and the RNG pruning strategy applied leads to the pruning of edge (u,v) by w. Consider a scenario where there exists a query point q such that $dist(u,q) < dist(v,q) < dist(w,q)$. During beam search on KNNG, v is found when u is included in *pool* as defined in Algorithm 1. However, after pruning, v may no longer be found even if u is included in *pool*. This is because w might not be successfully inserted in *pool* due to its longer distance. Thus, in such cases, the k-CNA quality on NSG is inferior to that on KNNG.

Notably, the RNG pruning is designed to guarantee the finding of the 1-NN through greedy routing when $q \in D$ [17]. Following this approach, several other pruning strategies have been introduced, e.g., [16, 48, 50]. However, to the best of our knowledge, there is currently no existing study that specifically addresses the pruning strategy for k-CNA beyond the context of finding the 1-NN.

Motivated by this, we introduce a novel pruning strategy named α-pruning for the refinement. Different to RNG pruning strategy, our strategy enables efficient retrieval of k-CNA results while allowing the control of the k-CNA quality by the parameter α.

α-pruning: An edge (u,v) exists in the graph only if there is no edge (u,w) in the graph where $dist(u,w) < dist(u,v)$, $dist(v,w) < dist(u,v)$, and $\angle uwv > \alpha$.

The practical effectiveness of our proposed α-pruning is demonstrated in the following example, with theoretical analysis to follow in the subsequent subsection.

Example 4.2: To compare the k-CNA results, we randomly sample query points from the dataset as queries, and conduct 50-ANN search with varying beam widths on Glove dataset using three different graph indexes: KNNG, NSG, and α-PG with $\alpha = 66°$. The accuracy of k-ANN search results is assessed from two angles: comparing results with the same beam width and comparing results within the same running time. In Figure 4, with the same beam width setting, it is evident that compared to KNNG, NSG compromises the quality of k-CNA results, whereas α-PG maintains similar or superior quality, especially with increasing beam width. From a time-based perspective, the search on α-PG emerges as the most effective choice.

4.2 Analysis of Our Scheme

In this subsection, we evaluate the performance of the *refinement-before-search* scheme by assessing the quality loss incurred by our pruning strategies in comparison to directly searching on KNNG.

Our discussion is based on comparing the search paths from the query to the ground-truth nodes on G and $\hat{G}$.

To be formal, we denote the KNNG as G and the one obtained by applying α-pruning on G as $\hat{G}$. For a k-CNA query q, we sort all nodes based on their distance to q, we assign them ranks denoted as $p_1, p_2, \ldots, p_n$, where the subscript represents the rank of each node, i.e., p_i is the i-NN of q and $rank(q,p_i) = i$. For a path $P = [v_1, \cdots, v_m]$, we denote $\delta(P) = \max_{x \in \{v_1,\cdots,v_m\}} rank(q,x)$, that is, $\delta(P)$ denotes the maximum rank of nodes among path P. For each node $u \in V(G)$, let $SP(q,u)$ be the set containing all paths from q to u in graph G, we use $\Delta(G,q,u)$ to denote the path in $SP(q,u)$ that minimizes the maximum rank among its nodes, i.e., $\Delta(G,q,u) = \arg\min_{P \in SP(q,u)} \delta(P)$.

Theorem 4.1: *Assume S_1, S_2 are the results of beam search with width L for query q on KNNG G and pruned KNNG $\hat{G}$ respectively. If $\delta(\Delta(G,q,p_k)) = \delta(\Delta(\hat{G},q,p_k))$, we have $p_k \in S_1$ implies $p_k \in S_2$.*

Proof Sketch: We assume $\Delta(G,q,p_k)$ and $\Delta(\hat{G},q,p_k)$ are distinct; otherwise, the proof is trivial. We prove it by contradiction. Assume $p_k \in S_1, p_k \notin S_2$ when $\delta(\Delta(G,q,p_k)) = \delta(\Delta(\hat{G},q,p_k))$. Then, there must exist a node $v \in \Delta(\hat{G},q,p_k)$ that has not been successfully inserted into the beam search queue, indicating that the path from q to p_k has not been discovered. Since $\Delta(\hat{G},q,p_k)$ exists in G and does not disturb the search on path $\Delta(G,q,p_k)$, we have $rank(v,q) < \delta(\Delta(\hat{G},q,p_k))$. However, for the same reason, such a node v does not exist, because the node with $\delta(\Delta(G,q,p_k))$ has been successfully inserted, not to mention a node like v with a lower rank. Hence, such v does not exist, which leads to a contradiction. □

According to Theorem 4.1, a node with a higher rank appearing in the search path on $\hat{G}$ indicates that we miss a closer neighbor in the k-CNA results, i.e., k-CNA quality loss compared with the search on G. Such a loss is caused by the α-pruning operations on G for the sake of k-CNA efficiency. Being a widely-used pruning strategy, we have demonstrated in Example 4.1 that RNG pruning results in higher ranks in the path, leading to quality loss. Next, our focus shifts to analyzing our newly proposed α-pruning strategy.

We employ the following lemma to establish the relationship between α and the k-CNA quality loss caused by α-pruning.

Lemma 4.1: *Assume the points are uniformly distributed in infinite space. If the RNG pruning strategy prunes an edge (u,v) caused by the edge (u,w), and $\angle uwv = \alpha$, then the probability that w has a higher rank than both u and v for any query point is $\frac{\pi-\alpha}{2\pi}$.*

Proof Sketch: Since the distance from w to query q is greater than the distance from u and v, it implies that q must be located in the

region divided by the perpendicular hyperplane between uw and vw and farthest from w. The angle corresponding to that region is $\pi - \alpha$. Therefore, there is a probability of $\frac{\pi-\alpha}{2\pi}$ that w has a higher rank than both u and v for query q. □

According to Lemma 4.1, as α increases, the probability that α-pruning leads to k-CNA quality loss decreases. As shown in Figure 6, through Monte Carlo simulations, the expected value of the angle $\angle uwv$ is approximately 100°. This indicates that one successful α-pruning leads to a higher maximum rank along the path (i.e., k-CNA quality loss) with about 0.2 ($\frac{180-100}{360}$) probability. Hence, we can control such a probability via α. Lemma 4.1 describes the relationship between α and k-CNA quality loss in the perspective of the neighborhood of a single node. In the following, we further present such a relationship in the view of the search path on $\hat{G}$.

Theorem 4.2: *Assume the points are uniformly distributed in infinite space. For any path $P = [q, \cdots, p_k]$, at least successfully utilizing α-pruning strategy $\frac{2\pi}{\pi-\alpha}$ times in expectation leads to a larger value $\delta(P)$ of the path.*

Proof Sketch: From lemma 4.1, we know the successful α-pruning has at most $Pr = \frac{\pi-\alpha}{2\pi}$ probability leads to a higher rank. Since each α-pruning is independent, the expected number of α-pruning leads to a higher rank is $1Pr + 2Pr(1-Pr) + 3Pr(1-Pr)^2 + \cdots = 1/Pr$. Hence, at least $\frac{2\pi}{\pi-\alpha}$ α-pruning in expectation leads to a higher rank in the path. □

According to Theorem 4.2, when α is small, a short search path from q to p_k will lead to k-CNA quality loss in expectation. For example, when $\alpha = 60°$, k-CNA quality loss happens once the path length reaches 3 in expectation. However, as we are aware, the path length from u to p_k is approximately 6 [33, 53]. Hence, this could result in quality loss when applying the RNG pruning strategy directly. Fortunately, through our proposed α-pruning strategy, we can mitigate the risk of quality loss in k-CNA by carefully setting the value of α. Note that when $\alpha = 60°$, α-pruning equals the RNG pruning. Hence, by setting $\alpha > 60°$, α-pruning is able to reduce the k-CNA quality loss caused by our *refinement-before-search* scheme compared with the RNG pruning. We will discuss the selection of α in the exp.1-b in Section 6.

5 A NEW PG CONSTRUCTION FRAMEWORK

In this section, we provide comprehensive details of our new construction framework for RNG and NSWG. Combining the approaches outlined in the previous section, we first present an optimized k-CNA approach in Section 5.1, and then we present our new construction methods for RNG on top of our optimized k-CNA approach in Section 5.2. Further, we enhance the NSWG construction by combining a layer-by-layer insertion strategy with the RNG construction framework in Section 5.3. In Section 5.4, optimization techniques are introduced to enhance the efficiency of RNG construction framework. Lastly, we consolidate all the methods discussed and present a streamlined and effective framework that can be applied to the construction of other PG methods, as in Section 5.5.

5.1 Optimized k-CNA Approach

In this part, we present the details of our k-CNA method following the *refinement-before-search* scheme proposed in the last section. We present OptKCNA in Algorithm 5. It takes the current k-CNA results $\{C(u)|u \in D\}$ as input, where $C(u)$ could be the k out-neighbors from the initial KNNG G_{k_0}, and outputs the refined k-CNA results by two steps, i.e. refinement and search. In refinement, a PG index $\hat{G}$ is directly derived by first applying α-pruning on G_{k_0} (Lines 1-4), where $C(u) = N_{G_{k_0}}(u)$, and then enhancing the graph connectivity (Lines 5-6). In search, the k-CNA results of each node u is enhanced via a k-ANN search on $\hat{G}$ (Lines 7-8).

Algorithm 5: OptKCNA($\{C(u)|u \in D\}, k, L, M, \alpha$)

Input : $\{C(u)|u \in D\}$ and four parameters k, L, m and α
Output : refined k-CNA results $\{C(u)|u \in D\}$
/* **Phase 2:** Refinement ($\hat{G}$ =Refine ($\{C(u)|u \in D\}, M, \alpha$)) */
1 **for** each $u \in D$ in parallel **do**
2 $N_{\hat{G}}(u) \leftarrow$ Prune $(u, C(u), \mathrm{M}, \alpha)$;
3 **for** each $u \in D$ in parallel **do**
4 $N_{\hat{G}}(u) \leftarrow$ Prune $(u, N_{\hat{G}}(u) \cup \{v|u \in N_{\hat{G}}(v)\}, M, \alpha)$;
5 find connected components via DFS;
6 add extra edges into $E(\hat{G})$ between connected components;
/* **Phase 3:** Search */
7 **for** each $u \in D$ in parallel **do**
8 $C(u) \leftarrow$ KANNSearch $(\hat{G}, u, k, L, u)$;
9 **return** $\{C(u)|u \in D\}$

Compared with the RNG construction, our method can be seen as a reversal of search and refinement. However, OptKCNA is more efficient due to the smaller node out-degrees and keeps the k-CNA quality with a high probability as our analysis in Section 4.2.

5.2 Self-Iterative Construction of RNG

According to our *refinement-before-search* scheme, we could generate the final RNG index by applying another refinement on the results of OptKCNA. However, such a simple construction strategy still faces the following challenges, in order to balance the k-CNA efficiency and quality.

Difficulty of Tuning Parameters in Search **Phase:** In the search phase, L is the key parameter to the efficiency-quality balance and thus should be carefully selected. However, the best choice of L varies from datasets. Grid search is commonly employed for finding the best L [33, 58], which builds a corresponding graph index for each potential L value with a huge cost. Hence, this one-shot strategy of setting L is considerably time-consuming.

To address this issue, we take the progressive strategy of tuning L and propose a self-iterative framework with two key components, i.e., **quality examination** and **iterative refinement**. The former determines the termination condition of our framework, while the latter defines the behaviour in each iteration. Again, we use the NSG as the representative RNG to illustrate the details.

Quality Examination: As determining the quality of NSG itself is challenging, we utilize the k-CNA quality to assess the quality of the derived NSG. However, computing the k-CNA quality can be time-consuming, as it requires brute-force computation of the ground truth (exact k-nearest neighbors of each node). To address this issue, we estimate the k-CNA quality via sampling. Specifically, we employ a random selection process to choose a specific number

Algorithm 6: IterNSG(D, k_0, k, L, M, α)

Input : dataset D and five parameters k_0, k, L, M and α
Output : an NSG G

/* **Phase 1:** Initialization */
1 build the initial KNNG G_{k_0} via KGraph [15];
2 **for** each $u \in D$ in parallel **do**
3 $C(u) = N_{G_{k_0}}(u)$;
/* **Phase 4:** Quality Examination */
4 **while** the estimator $\hat{r}$ does not achieve the requirement **do**
5 $\{C(u)|u \in D\} \leftarrow$ OptKCNA ($\{C(u)|u \in D\}, k, L, M, \alpha$);
6 estimate the quality of $\{C(u)|u \in D\}$ as $\hat{r}$;
/* **Phase 2:** Refinement */
7 G =Refine ($\{C(u)|u \in D\}, M$) as Lines 5-10 in Algorithm 2;
8 **return** G

Algorithm 7: OptHNSW(D, k_0, ef, M, α)

Input : dataset D and four parameters k_0, ef, M and α
Output : an HNSW G

1 **for** each $u \in D$ in parallel **do**
2 randomly determine the highest layer of u as $l(u)$;
3 $m_L \leftarrow \max_{u \in D} l(u)$;
4 randomly select ep from the points in layer m_L;
5 **for** $i \leftarrow m_L$ downto 0 in parallel **do**
6 $D_i \leftarrow \{u|l(u) \geq i\}$;
7 **if** $|D_i| \leq M$ **then**
8 $N_{G_i}(u) \leftarrow D_i$ **for** each $u \in D_i$;
9 **else**
10 $G_i \leftarrow$IterNSG ($D_i, k_0, ef, ef, M, \alpha$);
11 **return** $G = \{G_0, G_1, \ldots, G_{m_L}\}$

n_s of nodes (we will discuss later) and then compute the average recall over the k-CNA results of those sampled nodes as an estimator for the average recall over all nodes, i.e., the k-CNA quality.

We then delve into determining the value of n_s to ensure that the estimation obtained through random sampling possesses a theoretical guarantee. Let $r(u)$ denote the quality of current k-CNA results of u, and $r(G) = \sum_{u \in G} r(u)$ the precise sum of recall of each node in the graph G. Suppose we randomly select n_s nodes $S = \{u_1, \cdots, u_{n_s}\}$. We can compute their sum of recall, denoted as $\hat{r}(S) = \sum_{u \in S} r(u)$. By the Chernoff bounds [41], the following theorem proves $\frac{\hat{r}(S)}{n_s}$ is an accurate estimator of $\frac{r(G)}{n}$ when the number of samples n_s is sufficiently large.

Theorem 5.1: *Assume that n_s satisfies $n_s \geq (8+2\varepsilon) l \log n/\varepsilon^2$. Then the inequality $|\frac{\hat{r}(S)}{n_s} - \frac{r(G)}{n}| < \frac{\varepsilon}{2}$ holds with at least $1-n^{-l}$ probability.*

Proof Sketch: We regard $\hat{r}(S)$ as the sum of n_s i.i.d. Bernoulli variables with a mean $\mu = r(G)/n$. Then we have $Pr[|\frac{\hat{r}(S)}{n_s} - \frac{r(G)}{n}| \geq \frac{\varepsilon}{2}] = Pr[|\hat{r}(S) - n_s\mu| \geq \frac{\varepsilon n_s}{2}] = Pr[|\hat{r}(S) - n_s\mu| \geq \frac{\varepsilon}{2\mu} \cdot n_s\mu]$. Let $\delta = \frac{\varepsilon}{2\mu}$, by the Chernoff bounds, $\mu \leq 1$ and $n_s \geq \frac{(8+2\varepsilon) l \log n}{\varepsilon^2}$, we have $Pr[|\frac{\hat{r}(S)}{n_s} - \frac{r(G)}{n}| \geq \frac{\varepsilon}{2}] \leq exp(-\frac{\delta}{2+\delta} \cdot n_s\mu) = exp(-\frac{\varepsilon^2 n_s}{8\mu+2\varepsilon}) \leq \frac{1}{n^l}$. □

It is worth noting that such estimation can be executed asynchronously with index construction, allowing for the evaluation of the index after each iteration while progressively building the index. Simultaneously, we can estimate k-CNA quality and terminate the process once deemed satisfactory.

Iterative Refinement: Once we have the quality examination, we can design an iterative refinement approach for NSG construction. This involves conducting further searches on α-PG (i.e., the graph index derived by α-pruning on current k-CNA results with connectivity enhancement) when the quality is deemed insufficient. The details are presented in Algorithm 6. The initialization phase remains the same as the original NSG, where k-CNA results are obtained from a KNNG (Line 1-3). The iterative process continues until the quality requirement is met (Line 4). Alternatively, the termination condition can be set based on the number of iterations. Within each iteration, k-CNA results are refined using Algorithm 5 (Line 5). Once the iterative process terminates, the k-CNA results are further pruned via RNG pruning and connectivity enhancement to obtain the NSG (Line 7).

5.3 Global Construction of NSWG

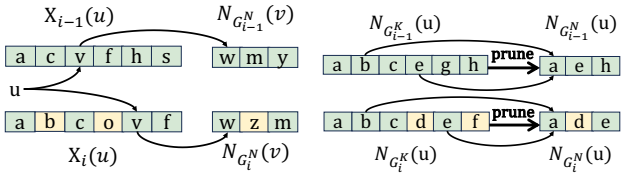
As mentioned in the previous section regarding the NSWG construction issue, the poor k-CNA quality primarily arises due to the incremental node-by-node insertion strategy and the search conducted on the current incomplete graph index that includes only a portion of nodes. In this part, we still use HNSW as the representative NSWG index, and propose a global construction of HNSW, which achieves k-CNA results on whole data points in each layer instead of only a part.

Our global construction of HNSW is built on top of a layer-by-layer insertion strategy in a top-down manner. To be specific, we first determine the layers for each node before the actual layer insertion and thus we obtain the whole set of nodes in each layer. Afterward, we employ our RNG construction framework in Section 5.2 to build a graph index for each layer. In this way, each layer of HNSW is built on top of k-CNA results w.r.t the whole set of nodes in each layer instead of only a subset. Hence, the k-CNA quality of each layer will significantly exceed 0.5 (i.e., the upper bound in the original HNSW). Note that each layer of HNSW is actually an RNG index, since it takes the RNG pruning strategy to prune the k-CNA results only without the connectivity enhancement via DFS. Subsequently, we insert the RNG index of the layer into HNSW according to the layer-by-layer insertion strategy.

The details of our HNSW construction are presented in Algorithm 7. Initially, we randomly determine the layer of each node, following the same approach as in the original HNSW (Lines 1-3). Then, we select one node at the top layer as the entry node ep (Line 4). The index construction proceeds from the top layer to the bottom layer (Line 5). For nodes in each layer (Line 6), we directly connect them if the number of nodes does not exceed the out-degree limit M (Lines 7-8). Alternatively, we apply our optimized NSG construction to obtain the index (Lines 9-10).

5.4 Implementation Details

In this part, we discuss two crucial optimization techniques aimed at improving the construction efficiency of our RNG framework as



(a) P2K (b) K2P

Figure 7: Examples of repeated distance computations

well as NSWG framework. To discuss the details briefly later, in the i-th iteration of Algorithm 6, k-CNA results are formed as a KNNG (of the dataset) denoted as G_i^K, and the α-PG is denoted as G_i^N.

P2K: from α-PG to KNNG. This process entails answering a k-ANN query on the current α-PG for each $u \in D$. The optimization efforts primarily avoid repeated distance computations. It is evident that multiple iterations of P2K involve repeated distance computations, as each point u is inquired multiple times, leading to redundant verification of its similar points. This happens during node expansions, as depicted in Lines 5-6 of Algorithm 1, when conducting k-ANN search for node u.

Example 5.1: Figure 7(a) shows an example, where the nodes colored yellow indicate new members in i-th iteration and $X_i(u)$ denotes the set of expanded nodes in the i-th iteration. In the example, the search path $u \rightarrow v \rightarrow w$ is found in two consecutive iterations and thus $dist(u, w)$ are computed twice.

In P2K, the computation of $dist(u, w)$ occurs in both the $(i-1)$-th and i-th iterations, when $v \in X_{i-1}(u) \cap X_i(u)$ and $w \in N_{G_{i-1}^N}(v) \cap N_{G_i^N}(v)$. In this case, v is expanded and w is a neighbor of v in both iterations. Furthermore, it is worth noting that for each node $v \in D$, $N_{G_i^N}(v)$ remains similar (with only a small portion being changed) when $N_{G_{i-1}^K}(v)$ is sufficiently accurate. As a result, the size of $N_{G_{i-1}^N}(v) \cap N_{G_i^N}(v)$ increases resulting in more repeated distance computations.

To address this issue, we propose adding additional information to the neighbor list of each node. Firstly, we compare $X_i(u)$ and $X_{i-1}(u)$ to determine whether each $v \in X_i(u)$ has already been in $X_{i-1}(u)$. We accomplish this by adding a boolean value to each member in $X_i(u)$. Secondly, we compare $N_{G_{i-1}^N}(v)$ and $N_{G_i^N}(v)$ to record whether each $w \in N_{G_i^N}(v)$ has appeared in $N_{G_{i-1}^N}(v)$. Similarly, we assign a boolean value to each member in $N_{G_i^N}(v)$. If a vertex $v \in X_i(u) \setminus X_{i-1}(u)$, we consider $N_{G_i^N}(v)$ as candidates of u. Otherwise, we only consider $N_{G_i^N}(v) \setminus N_{G_{i-1}^N}(v)$ as candidates. By implementing this approach, we can effectively reduce the number of repeated distance computations in consecutive iterations of P2K.

K2P: from KNNG to α-PG. We discuss the optimization of K2P. Like P2K, pruning operations for the same node are done in multiple iterations. Thus, there exist repeated distance computations and angle computations in consecutive iterations.

Example 5.2: As shown in Figure 7(b), let us consider $a \in N_{G_{i-1}^N}(u) \cap N_{G_i^N}(u)$ and $b \in N_{G_{i-1}^K}(u) \cap N_{G_i^K}(u)$. Notably, both $N_{G_i^N}(u)$ and $N_{G_i^K}(u)$ are sorted in ascending order of the distance from u. Hence, $dist(u, a) < dist(u, b)$ holds. In such case, $dist(a, b)$ and $\angle uab$ will be computed twice when pruning the edge set $\{(u, v) | v \in N_{G_{i-1}^K}(u))\}$ and $\{(u, v) | v \in N_{G_i^K}(u))\}$ respectively. That is because, when we check b in both iterations, a has been in $N_{G_i^N}(u)$ and we have to decide whether or not the edge (u, a) dominates (u, b).

Besides, as i increases, the quality of G_i^K improves, which results in an increase in unnecessary computations. Therefore, to reduce such repetition, we take a strategy like that in P2K. To be specific, we add extra information to distinguish whether or not each $b \in N_{G_i^K}(u)$ is also a member of $N_{G_{i-1}^K}(u)$ and each $a \in N_{G_i^N}(u)$ a member of $N_{G_{i-1}^N}(u)$. This could be easily implemented by adding an extra boolean value for each member in $N_{G_i^K}(u)$ and $N_{G_i^N}(u)$.

With such information, let us consider whether $b \in N_{G_i^K}(u)$ will join $N_{G_i^N}(u)$. If $b \notin N_{G_{i-1}^K}(u)$, we deal with it as the normal pruning, since there exist no distance computations between b and members in $N_{G_{i-1}^N}(u)$. Otherwise, we have $b \in N_{G_i^K}(u) \cap N_{G_{i-1}^K}(u)$. If $b \in N_{G_{i-1}^N}(u)$, we only check whether members in $N_{G_i^N}(u) \setminus N_{G_{i-1}^N}(u)$ but omit others in $N_{G_i^N}(u)$. Otherwise, we process it with a normal pruning. Hence, we can reduce those unnecessary computations without changing the final pruning results.

5.5 Summary

The pipeline of our framework is summarized in Figure 1(c). In the beginning, we obtain a KNNG through the initialization phase and then enter an iterative loop to continuously enhance the k-CNA quality until it satisfies the quality examination or achieves the required number of iterations. During each iteration, we first obtain an intermediate graph index from the current k-CNA results via a new pruning strategy for neighbor selection in the refinement phase. Subsequently, we perform a beam search for each node in the search phase to enhance the k-CNA quality. The optimization techniques, i.e., K2P and P2K, further accelerate the refinement and search phases respectively.

It is important to note that our framework is not only well-suited for NSG and HNSW, as detailed in this section, but also could be extended to the construction of other SOTA PGs, such as τ-MNG [48] and NSW [38] (as demonstrated in Exp.5 in Section 6). This is due to the fact that the construction pipelines of all SOTA PGs fall into the two categories we have discussed.

GPU Implementation. Our framework can be implemented on GPU. Firstly, in the initialization phase, we can utilize GNND [54], a GPU-based state-of-the-art KNNG construction method, to replace KGraph, as both GNND and KGraph are followed by the idea of NN-Descent [15]. Secondly, for the search phase, we can parallelize each search operation by treating each query independently. Thirdly, in the refinement phase, (i) since pruning on each vertex is independent, it can be efficiently parallelized; (ii) for connecting, its key operation is to detect connected components in the graph, which can be solved using existing GPU-based solutions, e.g., [3].

6 EXPERIMENTS

In this section, we present the results of our experimental study. We begin by introducing the experimental settings, followed by showcasing the building cost and search performance results when

Table 3: Data statistics

Dataset	size	#queries	dim.	type
Sift1M	1,000,000	10,000	128	Image
Gist1M	1,000,000	1,000	960	Image
Msong	992,272	200	420	Audio
Crawl	1,989,995	10,000	300	Text
Glove	1,183,514	10,000	100	Text
Deep1M	1,000,000	10,000	96	Image

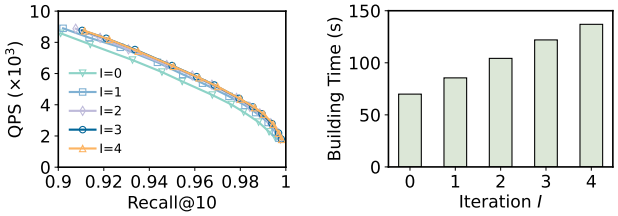
applying our framework to NSG, HNSW, τ-MNG and NSW. Furthermore, we compare our framework with four up-to-date methods to highlight the enhanced efficiency of our index construction without compromising search performance. Lastly, we analyze the effects of the optimization techniques employed and demonstrate the scalability of our approach on a large dataset.

Datasets: We use 6 public datasets with diverse sizes and dimensions. These datasets encompass a wide range of applications, including image (`Sift1M` [1], Deep1M [58] and `Gist1M` [1]), audio (`Msong` [11]) and text(`Crawl` [2] and Glove [49]). The statistics of those data sets are summarised in Table 3, where *#queries* denotes the number of queries and *dim.* denotes the dimensions of datasets. The query workloads of the datasets are given in the datasets. Besides, we take several random samples of distinct sizes from Sift50M [1] dataset to test the scalability of our methods.

Performance Indicators: Given a PG construction method, we care about two aspects of performance, i.e., construction cost and search performance. We use the execution time to evaluate the construction cost, denoted as "Building Time". For search performance, we care about efficiency measured by queries per second (QPS) and accuracy evaluated by recall. Given a query q, let k denote the number of returned neighbors, $N^*(q)$ be the exact k-nearest neighbors of q, while $N(q)$ be the set of k returned neighbors from different algorithms. The recall of the returned result is defined as $Recall@k = |N^*(q) \cap N(q)|/k$. We use the average recall over the query set to estimate the accuracy. We set the number k as 10 by default unless specified. All results are averaged over 5 runs.

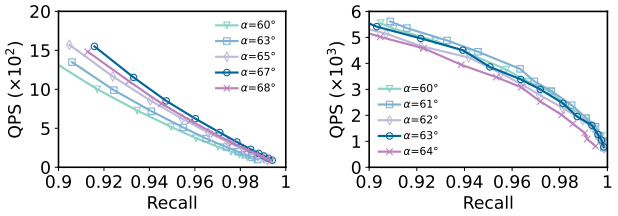
Computing Environment: All experiments are conducted on a server equipped with 2 Intel(R) Xeon(R) Silver 4210R CPUs, each of which has 10 cores (each supporting 2 hyper-threads), and 256 GB DRAM as the main memory. The OS version is CentOS 7.9.2009. All codes were written by C++ and compiled by g++ 11.3. The SIMD instructions are enabled to accelerate the distance computations.

Compared Algorithms: We mainly consider comparing the construction processes of two representative PG-based methods, i.e., NSG and HNSW, with our newly proposed framework. We refer to the original construction methods of NSG and HNSW as OriNSG and OriHNSW respectively. Additionally, we introduce our framework to enhance the construction of NSG and HNSW, denoted as FastNSG (Algorithm 6) and FastHNSW (Algorithm 7) respectively. We further compare our construction framework with three recently developed PG construction algorithms: DiskANN [50], LSH-APG [62], RNN-Descent [45] and ParlayANN [40]. In order to assess the generality of our framework, we conduct performance comparisons on τ-MNG [48] and NSW [38], both with (denoted as Fast τ-MNG and FastNSW) and without (denoted as Ori τ-MNG and OriNSW) the utilization of our proposed framework.



(a) QPS v.s. Recall (b) Building time

Figure 8: Effects of number of iterations on SIFT1M (Exp.1-a)



(a) Glove (b) Msong

Figure 9: Effects of value of α (Exp.1-b)

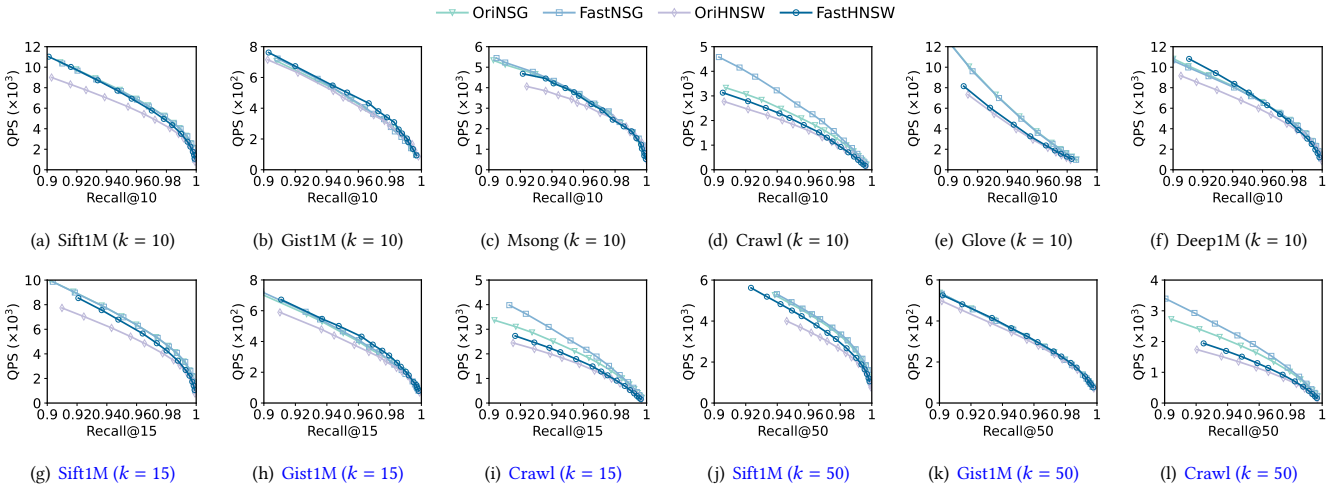
The index construction comparisons use all threads on the server, while the searches are compared using one single thread.

Exp.1: effects of the parameters. In the first experiment, we study the effects of the two newly proposed parameters in our approach: the number I of iterations and the value of angle α.

Exp.1-a: effects of I. In Algorithm 6, we iteratively conduct `K2P` and `P2K` before its termination. We use FastNSG as an example and show the result in Figure 8(a). We can see that more iterations improve the search performance but encounter marginal effects after the first two iterations. Hence, the result shows I affects the search performance of the finally derived proximity graph. Since the building time increases as I grows significantly as shown in Figure 8(b), we set I as 2 by default in later experiments.

Exp.1-b: effects of α. In this part, we study the effects of α on the k-CNA quality (measured by *recall*) and efficiency (measured by QPS). The results are shown in Figure 9. Notably, when $\alpha = 60°$, α-pruning equals to the RNG pruning. We can see that as α increases slightly, α-pruning enhances both the quality and efficiency of k-CNA compared with the RNG pruning. However, once α exceeds a specific value (e.g., 67° on Glove), the efficiency of k-CNA degrades, due to more out-neighbors left by α-pruning. Hence, α should be carefully tuned. In the following, we optimize α for each data via grid search, which starts from 60°.

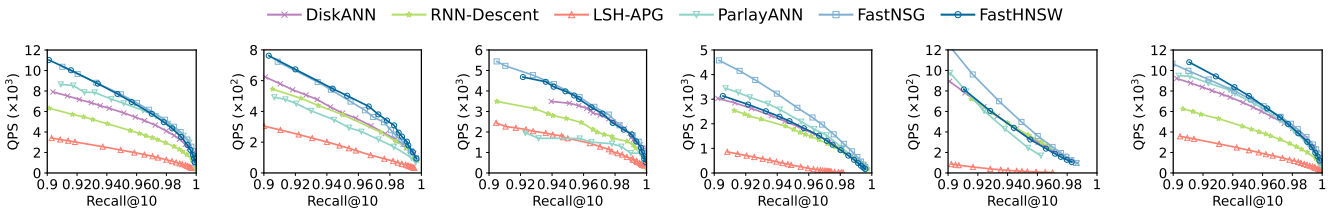
Exp.2 & 3: main experimental results in search performance & building time. In this part, we show the main results of this work by comparing our methods with the original algorithms in both search performance and building time. We carefully choose the construction parameters for each method. For OriNsg, we adopt the construction parameters provided by the authors in [17] for Sift1M and Gist1M datasets. As for the Msong, Glove and Crawl datasets, we utilize the parameters provided by the recent survey [58], which are determined using a grid search within the parameter space. We set the parameters of Deep1M as in [50]. For OriHNSW,



(a) Sift1M (k = 10) (b) Gist1M (k = 10) (c) Msong (k = 10) (d) Crawl (k = 10) (e) Glove (k = 10) (f) Deep1M (k = 10)

(g) Sift1M (k = 15) (h) Gist1M (k = 15) (i) Crawl (k = 15) (j) Sift1M (k = 50) (k) Gist1M (k = 50) (l) Crawl (k = 50)

Figure 10: Comparisons in search performance on NSG and HNSW (Exp.2)



(a) Sift1M (b) Gist1M (c) Msong (d) Crawl (e) Glove (f) Deep1M

Figure 11: Comparisons in search performance between existing approaches and ours (Exp.4)

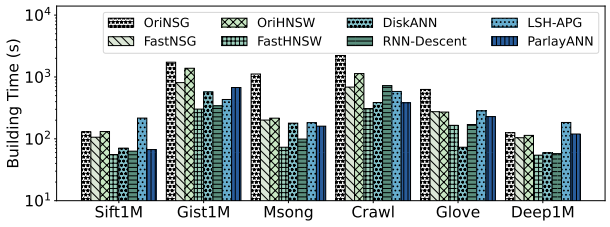


Figure 12: Comparisons in construction cost (Exp. 3 & Exp. 4)

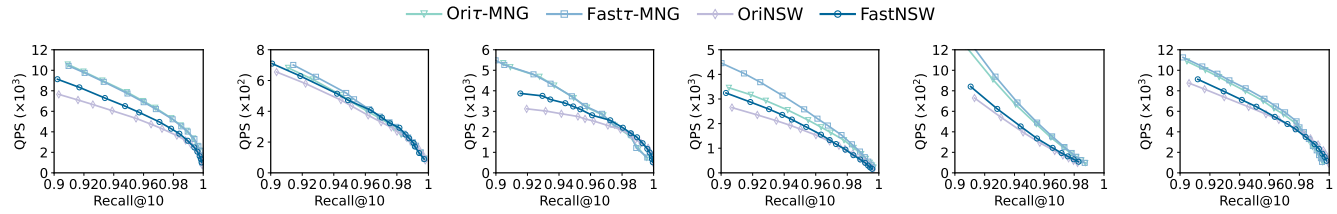
we set the value of M to be the same as in [58], and the value of ef is determined through a grid search within the parameter space.

The comparison of search performance and building time are presented in Figures 10 and 12, respectively. First, FastNSG exhibits comparable search performance to OriNSG in Figure 10, while significantly reducing construction costs as shown in Figure 12. Specifically, FastNSG achieves speedups of 1.2x, 2.1x, 5.6x, 3.2x, 2.3x and 1.2x over OriNSG on the Sift1M, Gist1M, Msong, Crawl, Glove and Deep1M datasets when we set k = 10, respectively. Second, FastHNSW shows significantly improved search performance compared to OriHNSW, primarily due to obtaining more accurate k-CNA results. Note that FastHNSW finds candidates for each point on the entire dataset in each layer, while OriHNSW only on a subset (the existing nodes in the graph during insertions). Compared with OriHNSW, FastHNSW accelerates construction by 2.4x, 4.6x, 3.0x, 3.7x, 1.6x and 2.1x speedups on the six datasets when k = 10.

Note that various approaches exhibit similar performance levels at high recall. This similarity arises from the fact that the search on proximity graph has two phases [60], and the diverse structures of different proximity graphs mainly affect performance on the first phase due to variations of short-distance neighbors, while having slight impacts during the second phase, i.e., when recall is high.

Exp.4: comparisons between existing approaches with ours. In this part, we compare our methods FastNSG and FastHNSW with other four recently proposed SOTA PG methods which focus on the index construction, i.e., DiskANN [50], LSH-APG [62], RNN-Descent [45] and ParlayANN [40]. We show the comparisons of search performance in Figure 11 and that of building cost in Figure 12. Overall, the results show that our methods achieve much less construction cost while obtaining better search performance.

Exp.5: extension of our framework on other SOTA PG approaches. In this part, we apply our framework to other two SOTA PG methods, i.e., τ-MNG (another SOTA RNG method) and NSW (another SOTA NSWG method). As shown in Figure 13, our method Fastτ-MNG achieves comparable or even better search performance compared with Oriτ-MNG, while FastNSW significantly achieves better search performance than OriNSW. Moreover, as in Figure 14, Fastτ-MNG achieves construction speedups of 1.2x, 16.4x, 5.7x, 3.3x, 2.3x, 3.1x and 2.1x over Oriτ-MNG on the six datasets respectively, while FastNSW obtains speedups of 4.6x, 3,1x, 3,6x, 1,6x, 2,1x and 2.1x respectively. Overall, our framework could be successfully applied to other SOTA PG methods, with superior construction efficiency and comparable search performance.



(a) Sift1M (b) Gist1M (c) Msong (d) Crawl (e) Glove (f) Deep1M

Figure 13: Comparisons of other SOTA PGs (Exp.5)

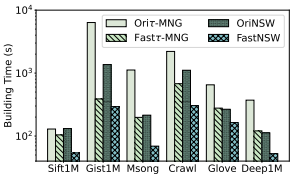


Figure 14: Construction time of other SOTA PGs (Exp.5)

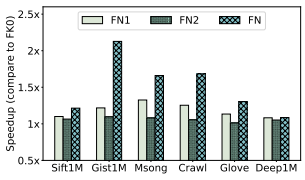


Figure 15: Effects of P2K and K2P techniques (Exp.6)

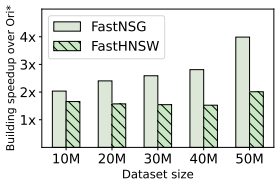


Figure 16: Scalability study of building index (Exp.7)

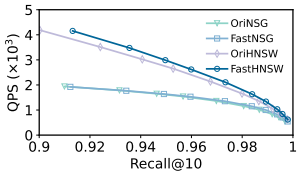


Figure 17: Scalability study of search on Sift50M (Exp.7)

Exp.6: effects of P2K and K2P techniques. Here, we delve into the effects of our P2K and K2P techniques for FastNSG. These techniques aim to reduce the redundant distance calculations of P2K in consecutive iterations (referred to as Opt1) and to reduce the redundant distance and angle computations of K2P in consecutive iterations (referred to as Opt2). These two strategies do not alter the final graph index, hence, we solely focus on the construction cost. We evaluate four approaches: FN0, which lacks any optimization; FN1, integrating solely Opt1; FN2, integrating solely Opt2; and FN, utilizing both two optimizations. The results are illustrated in Figure 15, where the time cost of each method is represented as the speedup over FN0. Overall, each optimization greatly speeds up FastNSG. These optimizations are independent of each other, with FN exhibiting the lowest construction cost. Besides, FN shows increasingly significant speedups as data dimensions increase, as both optimizations efficiently reduce the repeated computations.

Exp.7: scalability of our proposed methods. In this part, we assess the scalability of our methods on both index building and search performance using large data from Sift50M. As depicted in Figures 16 and 17, our Fast* approaches consistently accelerate over the original Ori* methods as dataset sizes grow without compromising the search performance. Notably, the speedup in FastNSG further amplifies with expanding dataset sizes, underscoring the exceptional scalability of our approach. This demonstrates our RNG construction framework scales well as the data size rises. We put the search performance across four additional scales of the Sift50M datasets in the full version on our GitHub repository.

7 RELATED WORKS

There have been a bulk of works on processing k-ANN queries on high-dimensional data in the literature. To answer a k-ANN query, index structures are widely used to carefully select a small part of high-quality candidates and then verify them via distance computations, in order to return accurate results with little cost. According to recent experimental studies [5, 32, 33, 58], proximity graphs [16, 17, 39] outperform other index structures such as hashing-based methods [14, 36, 37, 51], inverted index-based methods [7, 29] and tree-based methods [9, 42] in search performance. Due to their excellent search performance, the SOTA PG methods such as NSG [17] and HNSW [39] have been taken as the solution by industrial vector databases such as Milvus [55] and VBase [60].

Moreover, several works that combine the parallel power of GPU and the filtering capacity of PG methods have been proposed [21, 35, 46, 59, 61]. SONG [61] modifies the search method of existing methods such as HNSW in order to achieve higher throughput. Other GPU-accelerated methods such as GGNN [21], GANNS [59] and CAGRA [46] build their own graph index and have the corresponding search method. In addition, there exist I/O-efficient PG-based approaches, such as DiskANN [50] and Starling [57]. Besides, some work expands k-ANN search to various scenarios, such as hybrid search [19, 60, 63], out-of-distribution queries [27], and search on billion-scale datasets [13, 28, 43].

8 CONCLUSION

In this paper, we study the efficient construction of the PG-based approaches for k-ANN search. We first analyze the importance of k-CNA quality for PG index and identify their issues on index construction by revisiting existing PG construction approaches. To address these issues, we propose a novel construction framework for RNG with α-pruning strategy and a self-iterative framework. We then combine the layer-by-layer insertion strategy with our RNG construction framework to address the construction issue of HNSW (i.e., the representative NSWG). Extensive experiments are conducted on real-world datasets to show the superiority of our methods. The results show our approaches exhibit a construction speedup to 5.6x faster than the original methods of RNG and NSWG while delivering comparable or even superior search performance. For future work, it is a promising direction to integrate a more efficient KNNG construction method into our initialization phase for further enhancing construction efficiency.

REFERENCES

[1] 2010. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/.

[2] 2023. Common Crawl. https://commoncrawl.org/.

[3] Ghadeer Alabandi, William Sands, George Biros, and Martin Burtscher. 2023. A GPU Algorithm for Detecting Strongly Connected Components. In *SC*. ACM, 17:1–17:13.

[4] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. 2023. Retrieval-based Language Models and Applications. *ACL Tutorial* (2023).

[5] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.* 87 (2020).

[6] Franz Aurenhammer. 1991. Voronoi Diagrams - A Survey of a Fundamental Geometric Data Structure. *ACM Comput. Surv.* 23, 3 (1991), 345–405.

[7] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *IEEE TPAMI* 37, 6 (2014), 1247–1260.

[8] Kai Uwe Barthel, Nico Hezel, Konstantin Schall, and Klaus Jung. 2019. Real-time visual navigation in huge image sets using similarity graphs. In *Proceedings of the 27th ACM International Conference on Multimedia*. 2202–2204.

[9] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[10] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree: An Index Structure for High-Dimensional Data. In *VLDB*. 28–39.

[11] Thierry Bertin-Mahieux, Daniel P. W. Ellis, Brian Whitman, and Paul Lamere. 2011. The Million Song Dataset. In *Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR 2011*. University of Miami, 591–596.

[12] Jie Chen, Haw Ren Fang, and Yousef Saad. 2009. Fast approximate kNN graph construction for high dimensional data via recursive Lanczos bisection. *Journal of Machine Learning Research* 10, 9 (2009), 1989–2012.

[13] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search. In *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*.

[14] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*. 253–262.

[15] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*. 577–586.

[16] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE TPAMI* 44, 8 (2022), 4139–4150.

[17] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *PVLDB* 12, 5 (2019), 461–474.

[18] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality Sensitive Hashing Scheme Based on Dyanmic Collision Counting. In *SIGMOD*. 541–552.

[19] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *Proceedings of the ACM Web Conference 2023*. ACM, 3406–3416.

[20] Mihajlo Grbovic and Haibin Cheng. 2018. SIGKDD, Yike Guo and Faisal Farooq (Eds.). ACM, 311–320.

[21] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik PA Lensch. 2022. GGNN: Graph-based gpu nearest neighbor search. *IEEE Transactions on Big Data* 9, 1 (2022), 267–279.

[22] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In *ICML*. 3929–3938.

[23] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*. IEEE Computer Society, 5713–5722.

[24] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. [n. d.]. Embedding-based Retrieval in Facebook Search. In *KDD*. 2553–2561.

[25] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2016. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *PVLDB* 9, 1 (2016), 1–12.

[26] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2023. Atlas: Few-shot learning with retrieval augmented language models. *JMLR* 24, 251 (2023), 1–43.

[27] Shikhar Jaiswal, Ravishankar Krishnaswamy, Ankit Garg, Harsha Vardhan Simhadri, and Sheshansh Agrawal. 2022. OOD-DiskANN: Efficient and Scalable Graph ANNS for Out-of-Distribution Queries. *CoRR* abs/2211.12850 (2022).

[28] 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. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.

[29] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2011. Product quantization for nearest neighbor search. *IEEE TPAMI* 33(1) (2011), 117–128.

[30] Norio Katayama and Shin'ichi Satoh. 1997. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *SIGMOD*. 369 – 380.

[31] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. *NeurIPS* 33 (2020), 9459–9474.

[32] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. 2020. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020*. ACM, 2539–2554.

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

[34] Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, et al. 2024. RetrievalAttention: Accelerating Long-Context LLM Inference via Vector Retrieval. *arXiv preprint arXiv:2409.10516* (2024).

[35] Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, Chen Chen, Fan Yang, Yuqing Yang, and Lili Qiu. 2024. RetrievalAttention: Accelerating Long-Context LLM Inference via Vector Retrieval. arXiv:2409.10516 [cs.LG]

[36] Yingfan Liu, Jiangtao Cui, Zi Huang, Hui Li, and Hengtao shen. 2014. SK-LSH : an efficient index structure for approximate nearest neighbor search. *PVLDB* 7, 9 (2014), 745–756.

[37] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*. 950–961.

[38] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68.

[39] Yury Malkov and Dmitry Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE TPAMI* 42, 4 (2018), 824–836.

[40] Magdalen Dobson Manohar, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2024*. ACM, 270–285.

[41] Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized Algorithms*. Cambridge University Press.

[42] Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE TPAMI* 36, 11 (2014), 2227–2240.

[43] Jiongkang Ni, Xiaoliang Xu, Yuxiang Wang, Can Li, Jiajie Yao, Shihai Xiao, and Xuecang Zhang. 2023. DiskANN++: Efficient Page-based Search over Isomorphic Mapped Graph Index using Query-sensitivity Entry Vertex. *CoRR* abs/2310.00402 (2023).

[44] Shumpei Okura, Yukihiro Tagami, Shingo Ono, and Akira Tajima. 2017. Embedding-based News Recommendation for Millions of Users. In *SIGKDD*. ACM, 1933–1942.

[45] Naoki Ono and Yusuke Matsui. 2023. Relative NN-Descent: A Fast Index Construction for Graph-Based Approximate Nearest Neighbor Search. In *Proceedings of the 31st ACM International Conference on Multimedia, MM 2023*. ACM, 1659–1667.

[46] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2023. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. *arXiv:2308.15136* (2023). arXiv:2308.15136 [cs.DS]

[47] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Companion of the 2024 International Conference on Management of Data*. ACM, 597–604.

[48] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proc. ACM Manag. Data* 1, 1 (2023), 54:1–54:27.

[49] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, A meeting of SIGDAT, a Special Interest Group of the ACL*. ACL, 1532–1543.

[50] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishnaswamy, and Harsha Simhadri. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NeurIPS 2019*.

[51] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2015. SRS: Solving c-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *PVLDB* 8, 1 (2015), 1–12.

[52] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*. 563–576.

[53] Jeffrey Travers and Stanley Milgram. 1977. An experimental study of the small world problem. In *Social networks*. Elsevier, 179–197.

[54] Hui Wang, Wan-Lei Zhao, Xiangxiang Zeng, and Jianye Yang. 2021. Fast k-NN Graph Construction by GPU based NN-Descent. In *CIKM*. ACM, 1929–1938.

[55] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD '21: International Conference on Management of Data*. ACM, 2614–2627.

[56] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. 2017. A survey on learning to hash. *IEEE TPAMI* 40, 4 (2017), 769–790.

[57] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

[58] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *PVLDB* 14, 11 (2021), 1964–1978.

[59] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2022. GPU-accelerated proximity graph approximate nearest Neighbor search and construction. In *ICDE*. IEEE, 552–564.

[60] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023*. USENIX Association, 377–395.

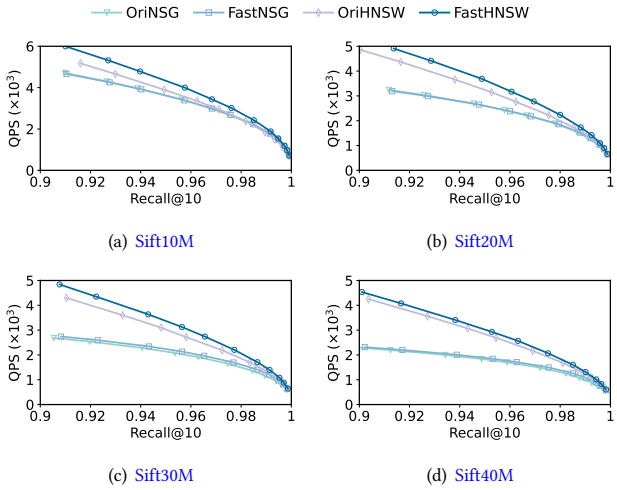
[61] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate nearest neighbor search on gpu. In *ICDE*. IEEE, 1033–1044.

[62] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards Efficient Index Construction and Approximate Nearest Neighbor Search in High-Dimensional Spaces. *PVLDB* 16, 8 (2023), 1979–1991.

[63] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 1 (2024), 69:1–69:26.

Appendix

ADDITIONAL EXPERIMENTS



(a) Sift10M (b) Sift20M

(c) Sift30M (d) Sift40M

Figure 18: Scalability study of search performance across four additional scales of Sift50M dataset (Exp. 7)

Exp.8: KNNG recall estimation via random sampling. In this experiment, we study the impact of the parameter ϵ on KNNG recall estimation (refer to Theorem 5.1). The results on the Sift1M dataset are shown in Figure 19, where the dotted line represents the exact value of recall@500 (the average recall of each node in KNNG). We find a reduction in running time as the value of ϵ increases. While the fluctuation range (error) of recall estimation expands with higher ϵ values, setting ϵ to 0.6 provides a reasonable estimate of recall swiftly, enabling a quick evaluation of KNNG quality.

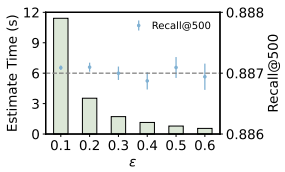


Figure 19: Recall estimation by random sampling (Exp.8)

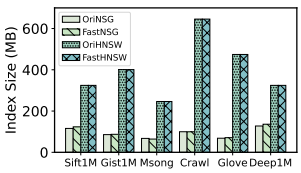


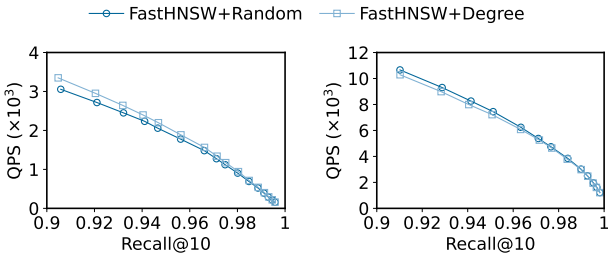
Figure 20: Index size comparisons (Exp.9)

Exp. 9: index size comparisons. We report the index sizes of our approaches compared to the original method of NSG and HNSW on all datasets, as depicted in Figure 20. The results indicate that our index closely resembles the original one, albeit slightly larger across all datasets.

Exp. 10: other strategies for the node selection of upper layers in HNSW. In previous HNSW comparisons, we follow the original HNSW method for the node selection of upper layers by randomly determining the nodes in each upper layer, which is denoted as FastHNSW+Random here. In this part, we explore the potential for a more refined approach to the node selection of each upper layer. However, given that the time required for such node selection is part of the index construction time, overly complex algorithms become impractical.

We introduce another strategy here, denoted as FastHNSW+Degree, which determines the nodes of upper layers in a bottom-up manner and iteratively selects nodes with maximum total out-degrees to the others in the current layer as the nodes of the next upper layer. We compare these two node selection strategies on Crawl and Deep1M datasets, with results detailed in Figure 21. FastHNSW+Degree enhances search performance on Crawl dataset but demonstrates inferior performance on Deep1M dataset. Additionally, FastHNSW+Degree demands more construction time, e.g., requiring an extra 2.8% and 8.0% building time on Crawl and Deep1M datasets respectively.

These results suggest the potential efficacy of alternative smarter strategies for node selection of upper layers in HNSW. However, it is vital to weigh such improvements against the added costs incurred during index construction. This highlights the need for further research to explore smarter node selection strategies for HNSW, and we leave this challenge as an open problem for future research.

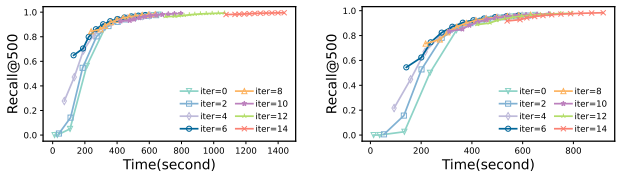


(a) Crawl (b) Deep1M

Figure 21: Other strategies for the node selection of upper layers in HNSW (Exp. 10)

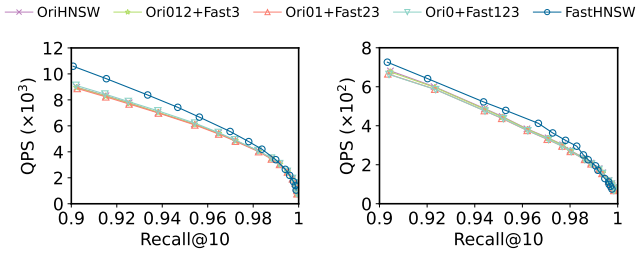
Exp. 11: incremental evaluation on HNSW by replacing layer by layer. In order to incrementally evaluate our layer placement strategies on HNSW, where each layer in HNSW is replaced by RNG, we replace each layer in HNSW with RNG for experimental study. Specifically, the HNSWs constructed on Sift1M and Gist1M datasets consist of 4 layers, ranging from layer 0 to layer 3, with layer 0 being the lowest layer and containing nodes of the complete datasets. As depicted in Figure 22, we begin by replacing layer 3 in HNSW, labeled as Ori012+Fast3, followed by the replacement of both layers 2 and 3, denoted as Ori01+Fast23, then we replace layers 1,2 and 3 in HNSW, denoted as Ori0+Fast123, and finally all layers in original HNSW are replaced, denoted as FastHNSW.

The results indicate that replacing higher layers has only a minimal impact on search performance, whereas replacing the lowest layer, layer 0, significantly enhances search performance. This finding aligns with our analysis of NSWG construction issue, where the presence of long-distance edges in the lower layers due to incremental insertion is unnecessary in HNSW.



(a) Gist1M (b) Crawl

Figure 23: The effects of KGraph construction cost on the quality of k-CNA results (Exp. 12)



(a) Sift1M (b) Gist1M

Figure 22: The search performance of the graph replacing layers of OriHNSW with that of FastHNSW (Exp. 11)

Exp.12: effects of the initial KNNG quality. To explore the effects of the initial KNNG quality on the final k-CNA results, we conduct additional experiments as follows. In these experiments, during the initialization phase, we change the number $iter$ of iterations of KGraph [15] to vary both its construction cost and quality. Specifically, when $iter = 0$, during initialization, k data points are randomly selected as neighbors for each data point. Next, when increasing the value of $iter$, at each iteration, every data point computed the distance between itself and its 2-hop neighbors to identify new, closer k data points as neighbors. Following this, our framework was employed to iteratively enhance the KNNG and conduct searches on the refined KNNG (i.e., PG) to generate a new KNNG.

The results are depicted in Figure 23, where the initial point of each line represents the time and quality of the KNNG derived from the initialization phase with different iterations. Each subsequent point along the line denotes the time and quality of the obtained KNNG at each iteration within our framework. The results illustrate that when striving for high-quality k-CNA results, e.g., the recall@500 is close to 1, the time remains steady across various settings of $iter$ for small $iter$ values, e.g., $iter \leq 8$. However, as $iter$ increases, the time required may significantly escalate compared to instances where $iter$ is small, e.g., $iter = 14$ in Gist1M.

Therefore, based on the above experimental analysis, it appears that constructing a high-quality KNNG in the initialization phase is not a compulsory step within our framework to achieve high-quality k-CNA results. However, there exists the potential to develop a more efficient method for constructing a high-quality KNNG that outperforms KGraph, the current state-of-the-art KNNG construction approach. Integrating such an approach into the initialization phase of our framework could further enhance the construction efficiency. This possibility is left open as a potential direction for future research.