HVS: Hierarchical Graph Structure Based on Voronoi Diagrams for Solving Approximate Nearest Neighbor Search

Kejing Lu
lu@db.is.i.nagoya-u.ac.jp
Nagoya University, Japan

Mineichi Kudo
mine@ist.hokudai.ac.jp
Hokkaido University, Japan

Chuan Xiao
chuanx@ist.osaka-u.ac.jp
Osaka University, Japan

Yoshiharu Ishikawa
ishikawa@i.nagoya-u.ac.jp
Nagoya University, Japan

ABSTRACT

Approximate nearest neighbor search (ANNS) is a fundamental problem that has a wide range of applications in information retrieval and data mining. Among state-of-the-art in-memory ANNS methods, graph-based methods have attracted particular interest owing to their superior efficiency and query accuracy. Most of these methods focus on the selection of edges to shorten the search path, but do not pay much attention to the computational cost at each hop. To reduce the cost, we propose a novel graph structure called HVS. HVS has a hierarchical structure of multiple layers that corresponds to a series of subspace divisions in a coarse-to-fine manner. In addition, we utilize a virtual Voronoi diagram in each layer to accelerate the search. By traversing Voronoi cells, HVS can reach the nearest neighbors of a given query efficiently, resulting in a reduction in the total search cost. Experiments confirm that HVS is superior to other state-of-the-art graph-based methods.

PVLDB Reference Format:
Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. HVS: Hierarchical Graph Structure Based on Voronoi Diagrams for Solving Approximate Nearest Neighbor Search. PVLDB, 15(2): 246-258, 2022.
doi:10.14778/3489496.3489506

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

1 INTRODUCTION

The approximate nearest neighbor search (ANNS) in a high dimensional Euclidean space is of great importance in applications such as databases, information retrieval, data mining, pattern recognition, and machine learning [1, 22]. The basic problem can be described as follows: given a dataset $\mathcal{D} \subseteq \mathbb{R}^d$ and a query $q \in \mathbb{R}^d$, the nearest object $x_* \in \mathcal{D}$, that is, $x_* = \arg\min_{x \in \mathcal{D}} \|x - q\|$, is expected to be found as fast as possible. Since its exact solution is hard to be solved efficiently because of the curse of dimensionality, many researchers turn to design approximate methods. Among various types of ANNS methods, graph-based methods show much better empirical search performance than others [14, 30]. Thus, it is timely to further reduce their search costs.

Let us first briefly review the working flow of existing graph-based methods. In the indexing phase, a proximity graph is constructed as the index structure in which the data points constitute the nodes and close node pairs constitute the edges. We can hop directly from one node to another only if an edge exists between them. Clearly, the search efficiency depends on the graph topology. Therefore, various edge-selection strategies have been proposed [14, 25, 30].

Although these graph-based methods have very different index structures, their search algorithms are quite similar. In every step, the closest point to the query in the priority queue is selected. Then, all its connected neighbors are visited and their exact distances to the query are computed. The priority queue is updated if closer points to the query are found. When no new point can be added into the priority queue, the search is terminated. Finally, the top K points in the priority queue are returned (for the K-ANNS problem).

It can be easily seen that if the size of the priority queue is 1, the standard search strategy incurs a computational cost of $O(\gamma v d)$, where γ is the length of the search path, v is the average out-degree of visited nodes, and d is the data dimension. In [14], the authors proposed a graph structure called NSG, which improves the search efficiency by reducing γ and v. However, their method does not reduce the $O(d)$ computational cost of the distance between every visited node and the query. In fact, exact distance computations are too costly and unnecessary in the early stage of searching. To overcome this limitation, we divide the search process into two stages. In the first stage, we approach a neighbor region of the query point, and then, in the second stage, we continue the search from this region to obtain the final results. Clearly, we can push exact distance computation to the later stage. This motivates us to devise a new graph structure to accelerate the search process.

In this paper, we construct Voronoi cells in such a way that every data point is contained in only one cell. Two data points are said to be close when they belong to the same Voronoi cell. Multiple Voronoi diagrams constitute a hierarchy in which the seed points (those points used to generate the Voronoi diagram) in a higher layer are also seed points in the lower layers. We assume such Voronoi diagrams, although in practice we use only their seed points so that these diagrams exist virtually. We call this index structure the *Hierarchical Voronoi structure (HVS)* (see Fig. 1 for the coarse-to-fine Voronoi diagrams). We also regard the HVS as a graph, or more

Proceedings of the VLDB Endowment, Vol. 15, No. 2 ISSN 2150-8097.
doi:10.14778/3489496.3489506

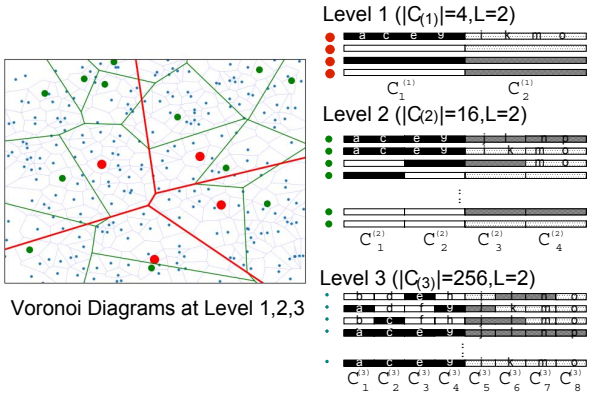


Figure 1: An example of HVS (Voronoi diagrams at three levels and the corresponding codebooks). On the right side, every $C_j^{(i)}$, i.e., the j-th sub-codebook at the i-th level, contains two elements ($L = 2$), which are represented by two rectangles of different shadows. Every seed point (graph node) is represented by a combination of elements in different sub-codebooks at the same level. At level-1, we can only generate four graph nodes marked in red. As the level increases, we can generate more graph nodes marked in green or blue by increasing the number of sub-codebooks. On the left side, the diagrams are simulated in 2D as conceptual diagrams such that the seed points at level t also appear at level $t + 1$, in the way of generation shown on the right side.

precisely, a hyper-graph consisting of multiple graphs. In a single layer, a Voronoi cell, or equivalently a seed point, corresponds to a node of the graph in the layer. After determining all the Voronoi cells in all the layers, we discard the cells containing no data point and then connect the remaining cells. The distance between two arbitrary cells in the same layer is defined as the distance between their seed points. Edges are formed between some pairs of close cells. It is clear that the selection of seed points plays a critical role in this process. In the next section, we will demonstrate the detailed seed-selection process.

In the query phase, starting from the top (coarsest) layer, we move from a layer of coarser-grained cells to the next lower layer of finer-grained cells. By means of a quantization technique, we find the closest cell to the query without any hop in the top layer, and this cell (node) becomes the entering point for the next layer. Then, stepping down one layer and starting from that entering point, we find the approximate nearest node in the layer. We repeat this process until the base layer is reached. In the base layer, because each original data point is a node in the graph, we conduct a standard graph-based search to obtain the final result.

In summary, our three-fold contributions are as follows:

(1) We propose a novel hierarchical structure called HVS to solve the ANN search problem. HVS consists of multiple graphs corresponding to multi-granular Voronoi diagrams. This structure accelerates the speed at which the approximate nearest neighbors of a given query are reached and thus significantly reduces the total searching time.

(2) By adopting a density-based allocation strategy, HVS reduces the number of data points appearing in the lower layers and thus reduces the indexing cost incurred by the points in the regions with low data density significantly.

(3) In the experiments, we combined HVS with the edge-selection strategy introduced in HNSW [30] to keep the high efficiency of indexing. Experiments confirmed that the proposed method improved the recall rates of existing graph-based methods by 3% - 10% with less indexing time.

2 RELATED WORK

Existing ANNS methods can be categorized into four classes: tree-based methods, LSH-based methods [19, 23, 27, 28, 34], quantization-based methods [10, 15, 20] and graph-based methods. These methods have different strengths. For examples, LSH-based methods and tree-based methods are suited to the datasets updated frequently while quantization-based methods generally have low memory footprints. In this paper, we focus only on graph-based methods since extensive experimental studies have shown their superiority on search performance among in-memory ANNS methods [11, 14, 25, 30]. For those scenarios in which the top priority is given to other performance metrics, as mentioned above, we recommend users to reasonably choose methods of other types. For the comprehensive surveys of existing in-memory or disk-based ANNS methods of these four types, see literatures [25] and [11].

2.1 Graph-based methods

Among graph-based methods, we particularly focus on two state-of-the-art ones, HNSW [30] and NSG [14]. Their working mechanisms are briefly introduced as follows.

Hierarchical navigating small world (HNSW) [30] uses a hierarchical structure built on a navigating small world graph [29]. The basic idea of HNSW is to construct multiple layers, on each of which only a part of data points in the dataset constitute the set of graph nodes. Specifically, every layer contains all nodes appearing in its higher layers and the base layer contains all data points. HNSW conducts the search in the direction from the top layer to the base layer and ensures that a locally nearest graph node can be found in each layer. The nearest point found in each layer is treated as the entering node in the next lower layer. Finally, HNSW searches the base layer by means of the standard search algorithm and returns the results from the priority queue.

Navigating spreading-out (NSG) [14] uses a single-layer graph structure. Specifically, NSG aims at the following four goals: (1) ensuring the connectivity of the graph; (2) lowering the average out-degree of nodes for fast traversal; (3) shortening the search path; and (4) reducing the index size. To this end, the authors in [14] proposed *Monotonic relative navigating graph* (MRNG) based on ideas of MSNET [2] and RNG [8]. Furthermore, the authors proposed a practical graph structure called NSG approximating MRNG. Since the out-degree of each node in NSG varies depending on data points (nodes), NSG can keep the high search efficiency of HNSW with a smaller index size.

Recently, inspired by the superior performance of HNSW, other graph-based methods and related techniques have been proposed for dealing with different scenarios. The authors in [32] focused

on the theoretical analysis of KNN graph in the scenario that dimension d is small ($d \leq 32$). Goldfinger [16] is a KNN graph built on Jaccard's index. SPTAG [7] is a tree-based and distributed ANN search method which supports GPU. GRIP [36] is a multi-core capacity-optimized multi-store ANN algorithm which focuses on the searching optimization in both DRAM and SSDs. In this paper, we focus on the standard scenario, that is, the CPU-based search in ℓ_2 metric, which is adopted by HNSW and NSG.

2.2 Why HNSW and NSG are appropriate benchmarks

Both HNSW and NSG are state-of-the-art methods. Specifically, compared with other graph-based methods, such as FANNG [18], nn-decent-based KGraph [6, 9, 17], IEH [21], Efanna [12], HNSW and NSG have shown their competitive search performances [13, 14, 30]. Moreover, they have been successfully implemented in the industry field. Recently, some researchers tried to introduce learning-based methods to improve the search performance of graphs [4, 35]. However, we need to pay attention to the following three points. (1) These methods require much more indexing time than HNSW, which is not suited to large-scale datasets. (2) Their complicated edge-selection strategies make insertions difficult. (3) HVS is orthogonal to most of learning-based methods.

For these reasons, we do not try to find the optimal edge-selection strategy for HVS since it highly depends on the data distribution. In addition, we do not consider those optimization techniques, such as [24], which are applicable to all compared methods. In this paper, we focus on the essential improvement brought by HVS.

2.3 Product quantization

We briefly introduce the basic idea of product quantization (PQ) [20] in a metric space as follows. Let $\mathcal{D} \subseteq E^d$ be a d-dimensional dataset and $q \in E^d$ be a query. We furthermore divide every point x in $\mathcal{D}$ into M sub-vectors such that $x = (x^{(1)}, x^{(2)}, \cdots, x^{(M)})$, where $x^{(j)} \in E^{d'}$ $(1 \leq j \leq M)$ with dimension $d' = d/M$, assuming d is divisible by M. In this way, PQ constructs M subspaces of dimension d', each of which contains the corresponding sub-vectors of all points. Then, in each subspace, every sub-vector is represented by one of L (=256) sub-codewords of length $k = \log_2 L$. The set of all sub-codewords in the j-th subspace is called a codebook and denoted by $C^{(j)}$. By concatenating the sub-codewords in all subspaces, an original point is represented by a codeword of length Mk which takes a value in $C^{(1)} \times \cdots \times C^{(M)}$.

2.4 Comparison with existing methods

For solving the ANNS problem, the authors in [10] proposed L&C which connects quantized vectors by HNSW. HVS is essentially different from *L&C* in the following three aspects: (1) **Different types.** L&C is a VQ-based method while HVS is a graph-based method in its own right. (2) **Different goals.** For *L&C*, the motivation of introducing HNSW is to reduce the candidate list efficiently at the expense of some loss of query accuracy for high target recalls, while the graph structure in HVS aims to approach the nearest neighbor of the query faster, which makes HVS consistently outperform other graph-based methods. (3) **Different technical details.** All core techniques of HVS, that is, multi-level quantization, density-based allocation and layer-to-layer connection introduced later are designed to achieve the goal of HVS mentioned above, while L&C takes a standard way of HNSW to connect quantized vectors.

Table 1: Notations

Notation	Explanation
$\mathcal{D}$	the d-dimensional dataset.
T	the level of HVS model.
V_t	the set of seed points of the Voronoi diagram at level t.
D_t	the set of data points at level t.
S_t	the set of representative seed points appearing at level t.
E_t	the set of inter-graph edges at level t.
δ	the parameter controlling $\lvert D_t \rvert$.
L	the size of every sub-codebook.
$C_i^{(t)}$	the i-th sub-codebook at level t.

3 THE CONSTRUCTION OF HVS

3.1 Basic ideas

The working flow of HVS is shown in Fig. 2 and a geometrical illustration of HVS is shown in Fig. 3 (Some notations used later can be found in Tab. 1). The space in every layer, except for the base layer, is partitioned into virtual Voronoi cells. For the top layer, those cells form the node set of a graph without edges. In the second layer or lower layers where the data size is decreased layer by layer, only high-density cells are connected to the next deeper Voronoi cell. Every eligible cell containing data points connects to other neighbor cells in the same layer and also connects to a cell in the next lower layer. Some of such cells are connected directly to data points in the base layer. In the query phase, stepping down the layers while finding the locally nearest cells to the query, we determine a group of entering points for the base layer. In the base layer, we take the standard search strategy and obtain the final results. The advantage of HVS over existing graph-based methods originates from the following two facts. (1) The distance computation of two Voronoi cells is much more efficient than that of two raw vectors. (2) HVS can search the base layer from multiple Voronoi cells returned from the upper layers to ensure the high query accuracy while both NSG and HNSW search the base layer from a single data point due to the limitations of their indexes.

3.2 HVS model

3.2.1 PQ and Voronoi diagram. Based on the idea of PQ [20], we first divide the original d-dimensional space, E^d, into 2 subspaces of $E^{d/2} \times E^{d/2}$, and then into subspaces of $E^{d/4} \times E^{d/4} \times E^{d/4} \times E^{d/4}$, and so on. Accordingly, a sample $\mathbf{x} \in E^d$ is divided into $(\mathbf{x}_1, \mathbf{x}_2)$, $(\mathbf{x}_{11}, \mathbf{x}_{12}, \mathbf{x}_{21}, \mathbf{x}_{22})$, and so on. As a result, at level t, we have 2^t subspaces of dimension $d_t = d/2^t$:

$$\mathbf{x} = (\mathbf{x}_{11\cdots1}, \mathbf{x}_{11\cdots2}, \ldots, \mathbf{x}_{22\cdots,2}) \quad \text{(the length of a suffix is } t\text{)}.$$

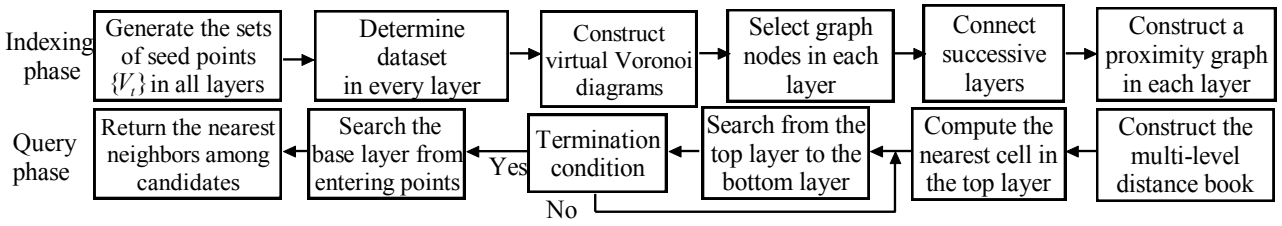


Figure 2: An overview of the working flow of HVS

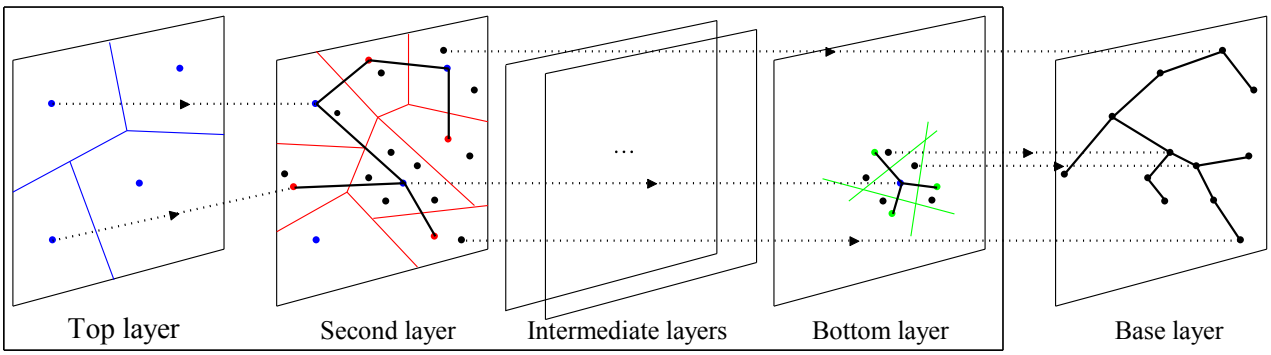


Figure 3: A geometrical illustration of HVS (see analysis in Sec. 3.1). Black points denote original data points and points in other colors denote seed points of corresponding Voronoi cells.

Then, at each level t, we construct a codebook $C^{(t)}$, which consists of 2^t sub-codebooks $C_j^{(t)} (j = 1, 2, \ldots, 2^t)$. Because we fix the number of sub-codewords to a constant L, we have $C^{(t)} = C_1^{(t)} \times C_2^{(t)} \times \cdots \times C_{2^t}^{(t)}$ including the L^{2^t} objective vectors in $\mathbb{R}^d$ generated by concatenating their sub-codewords in all possible combinations. These codewords in $C^{(t)}$ are called the *t-level codewords*.

In this paper, to facilitate understanding, we consider a Voronoi diagram virtually at each level. In practice, we regard the set V_t of L^{2^t} codewords as the set of seed points of the Voronoi diagram at level t. We construct a hierarchy of Voronoi diagrams such that

$$V_{t_0} \subset V_{t_0+2} \subset V_{t_0+3} \subset \cdots \subset V_{T-1} \subset V_T,$$

where t_0 is the level of the top layer, $t_0 + 2$ is the level of the second layer, $t_0 + 3$ is the level of the third layer, and so on. Noting that we skip level $t_0 + 1$ and the reason will be clear in Sec. 3.5.1. Here, this hierarchy is enforced by constructing a sub-vector $c_j \in \mathbb{R}^{d_t}$ at level t from two sub-codewords at level $t + 1$ as

$$c_j = (c_{k_j}, c_{\ell_j}),\ (c_{k_j}, c_{\ell_j} \in \mathbb{R}^{d_{t+1}},\ 2d_{t+1} = d_t)$$

while keeping the number of sub-codewords at L. This bottom-up construction preserves the inclusion relation (an example is given in Fig. 1, and the details are shown in Sec. 3.3).

In addition, from a given dataset $\mathcal{D}$, we construct a sequence of datasets $\{D_t\}$ as

$$D_{t_0+2} \supset D_{t_0+3} \supset \cdots \supset D_T,\ D_{T+1} = \mathcal{D}$$

, such that the number of data points decreases as t ($\geq t_0 + 2$) increases with the depth of the visited layers. We say that a point $\mathbf{x} \in \mathcal{D}$ is *terminated* in the layer at level t when $\mathbf{x}$ last appears in D_t and does not appear in the following D_{t+1}. We also call the layer at level t as the *terminal layer* of $\mathbf{x}$.

3.2.2 HVS. The HVS can be expressed as a hyper-graph, i.e., a sequence of graphs, denoted as

$$G = (G_{t_0}, G_{t_0+2}, G_{t_0+3}, \ldots, G_T, G_{T+1} = G_0, E),$$

where G_{t_0} is the graph in the top layer of level t_0, G_T is the graph in the bottom layer, G_0 is the graph in the base layer, and E is the set of inter-graph edges (Fig. 3).

In each graph $G_t = (V_t, D_t, S_t, E_t)$, V_t is the set of L^{2^t} seed points, D_t is the subset of $\mathcal{D}$, S_t is the subset of V_t, and $E_t \subseteq S_t \times S_t$ is the set of edges. E is the set of inter-graph edges ($E \subseteq \bigcup_{t<s} S_t \times S_s$). In the top layer, $G_{t_0} = (S_{t_0}, \emptyset, S_{t_0}, \emptyset)$, that is, neither the data point nor the edge exists inside G_{t_0}, which only contains nodes in S_{t_0} that are connected to a node in the second layer of level $t_0 + 2$. In the base layer, $G_{T+1} = G_0 = (\mathcal{D}, \mathcal{D}, \mathcal{D}, E_0)$; that is, all nodes are data points themselves, and they are connected to one another. In this study, G_0 is a proximity graph generated by a standard graph algorithm in [30]. The edge set E_0 is also obtained using this algorithm.

From the definition of V_t, we can observe that $|V_t| = L^{2^t}$ is extremely large when $t \geq 3$. However, we select only a small subset of V_t to form S_t. In fact, S_t contains only seed points with representatives, which are defined as follows:

Definition 3.1. Given a seed point at some level, we say that a data point is a *representative* of this seed point if the data point is closest to this seed point among all the data points assigned to this seed point.

Based on this definition, it is easy to see that the number of seed points having representatives in the layer of level t is not greater than $\min\{|D_t|, L^{2^t}\}$.

Next, we introduce two types of nodes used later.

Definition 3.2. We call a node *regular node* if it is the endpoint of some edge in a single layer. We call a node *initial node* if it is only the tail of some directed edge connecting successive two layers.

In the top layer of HVS, all nodes are initial nodes, while in the other layers, all nodes are regular nodes. The role of initial nodes is to determine the global entering point before searching the second layer of HVS. The details will be shown in Sec. 4.2.

In G_t ($t \geq t_0$), a graph node of V_t can be thought of as the seed point of a Voronoi cell. Then, from the monotonicity shown previously, we have

$$V_t \subset V_{t+1}, \text{ for } t \geq t_0 \text{ and } D_t \supset D_{t+1} \text{ for } t \geq t_0 + 2.$$

Therefore, as moving to deeper layers, we have denser Voronoi diagrams with lesser number of data points. This brings the efficiency in searching.

In the following sections, we will discuss the implementation details on the elements of G. To be specific, $\{V_t\}$ is discussed in Sec. 3.3 and Sec. 3.5.1; $\{D_t\}$ is discussed in Sec. 3.4; $\{S_t\}$ is discussed in Sec. 3.5.2; E is discussed in Sec. 3.5.3 and $\{E_t\}$ is discussed in Sec. 3.5.4.

3.3 Multi-level quantization technique

In this section, we introduce the multi-level quantization technique on the generation of V_t in each layer, and then explain how to select the graph nodes of S_t from them.

3.3.1 The selection of V_T. The sets $\{V_t\}$ are constructed layer by layer from bottom to top, that is, from V_T to V_{t_0}. Suppose that we have determined the value of T in the bottom layer. Then, we need to generate the set of T-level codewords. For this goal, we use PCA-OPQ [15] due to its better performance than traditional PQ. Specifically, we first transform the original data along the principal axes of the training data and construct subspaces by the allocation strategy introduced in [15]. Then we adopt OPQ [15] with 2^T sub-codebooks for the rotated training data and construct the sub-codebooks subspace by subspace. Finally, we need to store the related orthogonal rotation matrix.

3.3.2 From V_t to V_{t-1} ($t_0 < t \leq T$). In this step, we determine the set V_{t-1} of $(t-1)$-level codewords based on the set V_t of t-level codewords. As for the generation process of V_{t-1}, re-learning them in a similar way of V_T is clearly not applicable since we could not ensure that every $(t-1)$-level codeword is a t-level codeword, which spoils the inclusion relation of $\{V_t\}$ mentioned in Sec. 3.2. Therefore, instead of re-learning $(t-1)$-level codewords, we take a *merging and permutation* strategy to generate them. Since we have obtained 2^t sub-codewords at level t, each of which contains L sub-codewords, we choose two sub-codewords and couple them into one so as to obtain 2^{t-1} sub-codewords. Then, in each of coupled sub-codeword at t-level, we select L combinations from all L^2 possible combinations and concatenate them into sub-codewords of length $d_{t-1} = 2d_t$ (see Fig. 1 and Fig. 4). Clearly, in order to fulfill this idea, we need to consider (1) the way of coupling two sub-codewords, and (2) the way of choosing L sub-codewords (see Fig. 4).

In order to solve the first problem, we measure the relativity between the distributions of data points in the corresponding two subspaces in the same layer. First, we give the definition of concatenated sub-codeword as follows:

Definition 3.3. Given two sub-codebooks $C_i^{(t)}$ and $C_j^{(t)}$ at t-level, let $x = [x_{(1)}, \ldots, x_{(d_t)}]$ and $y = [y_{(1)}, \ldots, y_{(d_t)}]$ be two sub-codewords in $C_i^{(t)}$ and $C_j^{(t)}$, respectively. We call the concatenated sub-codeword $(x_{(1)}, \ldots, x_{(d_t)}, y_{(1)}, \ldots, y_{(d_t)})$ as the *concatenated sub-codeword* of x and y and denote it by $x \cdot y$, or shortly xy.

Next, we give the definition of the density functions of sub-codewords. Note that, in the searching process, we only hop among those seed points having representatives and have no knowledge on the locations of points assigned to them. Naturally, the number of points assigned to every sub-codeword is viewed as the density of this sub-codeword.

Definition 3.4. Given a sub-codebook C containing L t-level sub-codewords, the density of y in C, denoted by $f_C(y)$, is defined as the ratio of the number of data points falling inside the corresponding Voronoi cell of y to the total number n of data points.

Clearly, $f_C(y)$ is well-defined since $\sum_y f_C(y) = 1$. Then, we compute the mutual information of arbitrary two sub-codebooks C_i and C_j at t-level as follows (superscript t will be omitted whenever it is clear from the context).

$$I(C_i, C_j) = \sum_{x \in C_i, y \in C_j} f_{C_i \times C_j}(xy) \log\left(\frac{f_{C_i \times C_j}(xy)}{f_{C_i}(x) f_{C_j}(y)}\right), \quad (1)$$

where $f_{C_i \times C_j}(xy)$ denotes the density of concatenated codeword xy. Since it is shown in [33] that product quantization is more effective for the data distribution with high entropy, we merge those pairs of sub-codebooks with high mutual information. Since the way of coupling is determined, we next explain how to choose two codebooks $C_i^{(t)}$ and $C_j^{(t)}$. By σ, let us denote a permutation of $\{1, \ldots, 2^t\}$. Here we limit σ to be a product of 2^{t-1} transpositions such that $\sigma = (a, b)(c, d), \cdots, (e, f)$, where all symbols are distinct. Among such $\sigma's$, we choose σ by the following equation.

$$\sigma = \arg\max_{\sigma} \sum_{i=1}^{2^{t-1}} I(C_{\sigma(2i-1)}^{(t)}, C_{\sigma(2i)}^{(t)}) \quad (2)$$

That is, the i-th sub-codebook at level $t-1$ combines the $\sigma(2i-1)$-th and the $\sigma(2i)$-th sub-codebooks at t-level.

If we consider every codebook as a node and their mutual information as the weight of the edge connecting them, (2) becomes an NP-hard graph-partition problem. Therefore, we select greedily the edge with the largest weight in every step until all codebooks are coupled, which works well enough for our proposal.

After determining the permutation σ, we use an orthogonal matrix R_t, which is defined as follows, to represent the permutation of sub-codebooks at t-level:

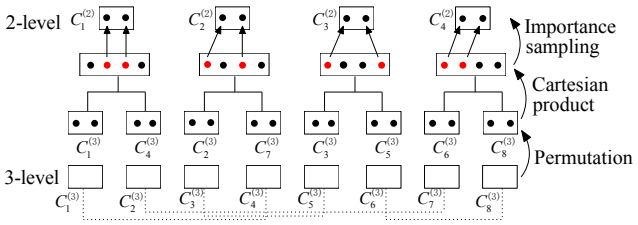


Figure 4: An illustrative explanation of merging and permutation strategy. Here, we show the generation process of 2-level codewords based on the 3-level codewords. In this example, the number of sub-codewords in every sub-codebook is $L = 2$. $C_i^{(t)}$ denotes the i-th sub-codebook at level t; black points denote sub-codewords in the sub-codebook and red points denote the selected 2-level sub-codewords in the concatenated sub-codebook.

$$R_t = \begin{bmatrix} 0 & \cdots & I_{\sigma^{-1}(1)} & \cdots \\ 0 & I_{\sigma^{-1}(2)} & & 0 \\ & \vdots & \ddots & \\ 0 & \cdots & I_{\sigma^{-1}(2^t)} & \cdots \end{bmatrix} \in E^{d \times d} \tag{3}$$

Here, $I_{\sigma^{-1}(i)}$ denotes the $\sigma^{-1}(i)$-th identity matrix of size d_t. By the feature of multi-level quantization technique introduced above, it is easy to see that we only need to use a single rotation matrix $R = R_0 \times \prod_{t=t_0+2}^{T} R_t$ to represent the ways of permutations at all levels, where R_0 denotes the rotation matrix with respect to PCA-OPQ in the bottom layer, since the way of coupling sub-codebooks at t-level has no effect on that at $t+1$ and higher levels.

Next, let us talk about the second problem of choosing L combinations. In $C_j^{(t)} \times C_k^{(t)}$, we obtain L^2 possible concatenated sub-codewords, and choose L sub-codewords from them to form an L-coreset. First, we generate L centers by weighted k-means and then moving them to their nearest concatenated sub-codewords. Since there may exist duplicate sub-codewords after moving centers, we also adopt the importance sampling technique introduced in [3] to generate the rest sub-codewords. This is because (1) such a constructed coreset is a k-means coreset, which accords with our classification strategy based on codewords, and (2) the generation process is easy-to-implement and fast.

3.3.3 The selection of V_{t_0}. Since all seed points in the top layer are initial nodes, the way of selecting them is a little different. We choose $t_0 = 2$ since the total number of 0-level or 1-level codewords is too small, while $|V_{t_0}| = L^4$. The details are as follows. In the sampling step of 2-level, we only choose $L = 16$ sub-codewords in each concatenated sub-codeword and obtain 16^4 codewords at 2-level. We treat them all as the seed points of the Voronoi cells in the top layer. Later, in Sec. 4, we will show how to find efficiently the Voronoi cell in which the query falls. The seed point of such a Voronoi cell will be treated as the global entering point of HVS.

Algorithm 1: Indexing of HVS

Input: $\mathcal{D}$ is the dataset; T is the level number of bottom layer; δ ($\delta < 1$) is the coefficient controlling the data size in each layer;

Output: Graph structure G; $\{C_i^{(T)}\}_{i=1}^{2^T}$ for the bottom layer; permutation matrix R; indexes of sub-codewords $\{I_{i,j}^{(t)}\}$ ($t_0 \le t \le T-1$)

1 Determine $\{V_t\}$ associated with $\{C_i^{(T)}\}_{i=1}^{2^T}$, R and $\{I_{i,j}^{(t)}\}$ by multi-level quantization technique;
2 Determine $\{D_t\}$ by density-based allocation strategy;
3 **for** *each t from t_0+2 to T* **do**
4 Assign every point in D_t to its nearest seed point in V_t;
5 Discard empty Voronoi cells and obtain S_t;
6 Add inter-graph edges by the strategy in Sec. 3.5.3;
7 Add the set of edges E_t into G_t ($t_0+2 \le t \le T+1$) by the existing edge selection strategy.

3.4 Density-based data allocation strategy

3.4.1 Motivation. Next, we focus on the determination of $\{D_t\}$. Since the space partition becomes finer as the layer goes deeper, it is natural to ask the following question: how do we judge if the Voronoi cell in which point x falls in some layer has distinguished x from other points accurately enough? If so, x does not need to appear in the lower layers for saving space. To answer this question, we make a judgement for every point x based on the following two factors: (1) the data density around x, and (2) the quantization error of x to its image. Here, the first factor is critical since the searching in the region with high data density is hard, which requires coarse-grained Voronoi cells to distinguish the points insides. In Sec. 3.4.2, we will present a criterion taking both of these two factors into consideration.

3.4.2 Criterion. We suppose that every data point x is sampled from a bounded space $S \subset E^d$ according to the p.d.f $f(x)$. Then, given all codewords in some layer, the overall quantization error $Q_n(S)$ is defined as follows:

$$Q_n(S) = \mathbb{E}_S[Q_n(x)] = \int_S f(x)Q_n(x)dx = \int_S g_n(x)dx, \tag{4}$$

where $Q_n(x)$ is the quantization error of x and $g_n(x) = f(x)Q_n(x)$. Clearly, in order to control the value of $Q_n(S)$, we only need to bound $g_n(x)$. Since the real value of $g_n(x)$ is unknown, we estimate the value of $g_n(x)$ by the k-nearest neighbor density estimate $\hat{f}_n(x)$ in [5]:

$$\hat{f}_n(x) = \frac{k}{nV_d \left\| X_k(x) - x \right\|^d}. \tag{5}$$

Here, n is the data size; $V_d = \pi^{d/2}/\Gamma(1+\frac{d}{2})$ and $X_k(x)$ is the k-th nearest neighbor of x. Let $\{x_1, x_2, \ldots, x_n\}$ be the original dataset and $\hat{g}_n(x) = \hat{f}_n(x)Q_n(x)$. Then, we estimate $\int_S g_n(x)dx$ by $\int_S \hat{g}_n(x)dx$ and by the result in [5] (p. 37).

Algorithm 2: Searching in HVS

Input: $\mathcal{D}$ is the dataset; q is query; T is the level number of bottom layer; $\{C_i^{(T)}\}_{i=1}^{2^T}$ are sub-codebooks for the bottom layer; R is the permutation matrix; K is the number of returned points;

Output: K approximate nearest neighbors

Let priority queue P be empty;
Rotate q to Rq and generate the multi-level distance book level by level;
Compute the nearest seed point of q in the top layer and insert its neighbors into P;
for *each* i *from* t_0+2 *to* T **do**
  for *every unvisited graph node* x *in the* i*-level layer* **do**
    Access its neighbors and compute distances;
    Update P if necessary;
  if *(*$i+1$*)-level layer exists* **then**
    for *every node* x *in* P **do**
      if x *is not terminated at* i*-level* **then**
        Take its connected node x' at $(i+1)$-level and compute the distance of x' to q;
        Update P if necessary;
Treat representatives of returned Voronoi cells in P as entering points in the base layer;
Search the base layer by the standard search strategy;
Return K nearest neighbors in P;

Algorithm 3: Insertion in HVS ($T = 4$)

Input: G is the current graph; q is data point to be inserted;

Compute multi-level distances of q to seed points and determine the entering point of q in the top layer;
Search the HVS layer;
if *there is no point assigned to* q*'s codeword* **then**
  Select neighbors from P and connect q with them;
else if q *is closer to the objective vector than the current representative* q' **then**
  Replace q' by q;
Start from neighbors of q in the HVS layer and turn to search the base layer;
Select neighbors from P and connect q with them;

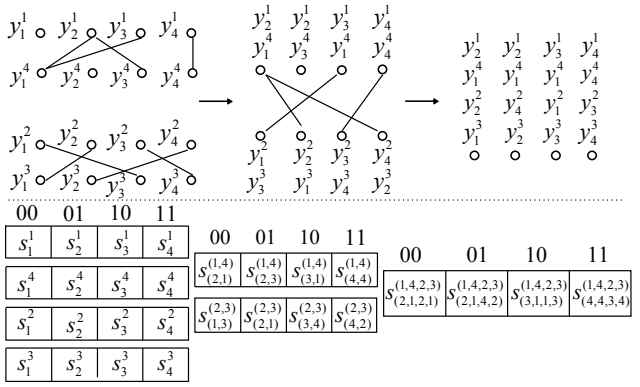


Figure 5: An example of the generation of multi-level distance book. For ease of presentation, we only show the process from 2-level to 0-level and suppose $L = 4$. Here, y_j^i denotes the j-th sub-codeword in the i-th sub-codebook at 2-level.

Lemma 3.5. *Assume that $f(x)$ is uniformly continuous. If $k/\log n \to \infty$ and, additionally, $k/n \to 0$, we have*

$$\int_S \hat{g}_n(x)dx \to \int_S g_n(x)dx \quad as\ n \to \infty. \tag{6}$$

Based on Lemma 3.5, we determine $\{D_t\}$ from the second layer to the bottom layer. Let U_t be a threshold for $g_n^{(t)}(x) = \hat{f}_n(x)Q_n^{(t)}(x)$, where $Q_n^{(t)}(x)$ is the quantization error of x appearing at t-level ($t \geq t_0+2$). If $g_n^{(t)}(x) > U_t$, we keep x at $(t+1)$-level and continue a similar judgement for the lower layers until reaching the bottom layer. Otherwise, x is terminated. That is, x is included in $D_{t_0+2}, D_{t_0+3}, \ldots, D_t$, where the layer of level t is the terminal layer of x.

3.4.3 Details. In the procedure above, we need to specify the values of $\{U_t\}$ such that $D_t \subset D_{t-1}$ for $t_0+3 \leq t \leq T$, but it is difficult in practice. Therefore, we take the following practical strategy to determine $\{D_t\}$. Suppose that D_{t-1} has already been determined. Then, we sort all data points $x \in D_{t-1}$ in the descending order of $\hat{f}(x)Q_n^{(t-1)}(x)$ and choose the first $\delta\,|D_{t-1}|$ points, where $\delta < 1$ is a user-specified coefficient. Clearly, such a strategy not only makes $\{D_t\}$ nested, but also brings an exponential decrease of the size of D_t. In fact, it is easy to see that, except for at most $n\delta^{T-t_0-2}$ points in the bottom layer, the minimum quantization errors $\min_t Q_n^{(t)}(x)$ of other points x's are bounded by $\max_t\{U_t\}$.

3.5 Implementation and algorithm

3.5.1 Determination of levels. We start from $t_0 = 2$ to ensure a reasonable number of seed points ($|V_{t_0}| = L^4$). Then skipping 3-level, we choose 4-level for the second layer. The reason of skipping 3-level is that we can approximate most of data points very well with L^{16} codewords ($L = 256$) in practice, as shown in various PQ based methods [15, 20, 33]. As for parameter T, we will discuss its setting in Sec. 6.1.2.

3.5.2 Construction of virtual Voronoi diagram. We have already given the set of Voronoi seed points at t-level. In each Voronoi cell, we find the nearest data point in D_t to the seed point as the representative of the cell. By gathering the seed points whose Voronoi cells have representatives, we obtain the sets of graph nodes $\{S_t\}$.

3.5.3 Layer-to-layer connection. After determining $\{S_t\}$, we determine the set E of inter-graph edges. For every eligible Voronoi cell u containing a seed point in S_t ($t \geq t_0+2$), we judge whether or not to connect u with a Voronoi cell $w \in S_{t+1}$ by the following criterion. **Case 1:** if t-level is the terminal layer for all points in u, we

connect u to its representative $x_u \in \mathcal{D}$ in the base layer (see Fig. 3).
Case 2: otherwise, we choose the point x'_u, that is not terminated at t-level and closest to the seed point, and connect u to the seed point whose Voronoi cell $w \in S_{t+1}$ contains x'_u (it exists because x'_u is not terminated at level t).

In the top layer, since every 2-level codeword appears as an independent graph node, we treat it as a query and find its k_0 approximate neighbors in the second layer by the existing graph-based method. Then, we connect every seed point to its k_0 neighbors by directed edges, which makes a connection between the top layer and second layer. This process can be conducted efficiently since every 2-level codeword in the top layer can also be viewed as a 4-level codeword in the second layer. In our experiments, k_0 is fixed to 10 on each dataset.

3.5.4 Indexing algorithm. The remaining work is to connect graph nodes in the same layer. For this purpose, we adopt the same edge selection strategy of HNSW [30]. Specifically, we insert all graph nodes sequentially into a graph initialized by a single node. For every graph node p to be inserted, we conduct the KNN search and find a group of neighbors of p. Then we choose some nodes from these neighbors and connect p with them. For those selected nodes, we also need to adjust their neighbors if necessary. By using such strategy, our proposed method can support insertions (see Sec. 5.2).

Now, we are ready to show the complete indexing algorithm of HVS (Algorithm 1). In Step 1, except for the bottom layer, we record $I = \{I_{i,j}^{(t)}\}$ ($t < T$, $1 \le i \le 2^t$), which indicates $id(x)$ and $id(y)$ as well as $\sigma(2i-1)$ and $\sigma(2i)$ for the j-th sub-codeword xy in the i-th subspace at t-level.

4 THE SEARCHING IN HVS

4.1 Multi-level distance book

For a given query q, our first step is to construct the multi-level distance book, which stores the distances of query q to the sub-vectors at each level. It is constructed level by level from T-level to t_0-level (Fig. 5). Let y_j^i be the j-th sub-codeword in the i-th sub-codebook at T-level and $s_{(b_1,\ldots,b_{J_t})}^{(a_1,\ldots,a_{J_t})}$ be the distance between q and an arbitrary sub-codeword $y_{b_1}^{a_1} \wedge \ldots \wedge y_{b_{J_t}}^{a_{J_t}}$ at t-level, where $J_t = 2^{T-t}$. Clearly, $s_{(b_1,\ldots,b_{J_t})}^{(a_1,\ldots,a_{J_t})}$ can be computed as follows:

$$s_{(b_1,\ldots,b_{J_t})}^{(a_1,\ldots,a_{J_t})} = \sum_{i=1}^{J_t} \left\| q^{(a_i)} - y_{b_i}^{a_i} \right\|, \tag{7}$$

where $q^{(a_i)}$ is the a_i-th sub-vector of q. Thanks to the multi-level quantization, the values of $s_{(b_1,\ldots,b_{J_t})}^{(a_1,\ldots,a_{J_t})}$ in the layer of level t can also be computed as follows.

$$s_{(b_1,\ldots,b_{J_t})}^{(a_1,\ldots,a_{J_t})} = s_{(b_1,\ldots,b_{J_{t-1}})}^{(a_1,\ldots,a_{J_{t-1}})} + s_{(b_{J_{t-1}+1},\ldots,b_{J_t})}^{(a_{J_{t-1}+1},\ldots,a_{J_t})} \tag{8}$$

In other words, we just compute the exact distances between quantized vectors and the query in a standard way (equation (7)) at T-level. For other levels, we use (8) requiring only additions to compute the distances efficiently.

4.2 Searching algorithm

The query phase is shown in Algorithm 2. Given query q, after obtaining the multi-level distance book (step 2), we can find the nearest sub-vector y^i of sub-vector $q^{(i)}$ ($1 \le i \le 4$) in the top layer, since each sub-codebook only contains L (=256) sub-codewords. Clearly the closest seed point (y^1, y^2, y^3, y^4) of q in the top layer is treated as the global entering point (step 3). From step 4, we conduct the search layer by layer. In each layer, we conduct the standard searching algorithm introduced in Sec. 1 and reach a group of approximate nearest Voronoi cells along the internal edges in this layer (steps 5-7). For each reached Voronoi cell, we get its representative and determine the Voronoi cell in which this representative falls in the next layer by the edges between successive layers. Naturally, such a cell is an entering point in the next layer (step 13). During this process, HVS may terminate the search in some middle layer, depending on whether or not the current layer is the terminal layer of the locally nearest Voronoi cell. If so, we stop the following hops from this cell and treat the representative of this cell as the entering point in the base layer (steps 8-12). Finally, we can obtain a group of entering points in the base layer and then execute the standard algorithm to obtain the final results (step 14).

5 DISCUSSION

5.1 Complexity analysis

Since HVS adopts the same edge selection strategy of HNSW [14], the following complexity analysis is based on this fact. In addition, we assume that $\delta \le 0.5$.

Time complexity. In [30], if the size of priority queue is set to 1, the authors estimate the searching complexity of HNSW as $O(vd \log n)$ under some assumptions, where n is the data size; d is the dimension and v is the maximum out-degree. Based on this result and the fact that every t-level codeword is essentially a d-dimensional vector, the searching complexity of HVS is estimated as $O(2Mv_0 \log n + v_1 d \log n')$, where $M = 2^T$ is the number of sub-codebooks in the bottom layer; n' is the number of points falling inside the ball-like query-centric region before searching the base layer (due to the feature of graph-based searching, this region may shrink in the base layer when closer points are found); v_0 is the average out-degree of seed point and v_1 is the average out-degree of data point. Intuitively speaking, $2 \log n$ and $\log n'$ are the estimated lengths of search paths in upper layers and in the base layer, while Mv_0 and $v_1 d$ are the computational costs in upper layers and in the base layer. Combining the observation that n' is generally less than $10^{-5}n$ on real datasets and the fact that $M \ll d$ for high-dimensional datasets, we can see that, generally, HVS has the lower time complexity than that of HNSW in practice.

Space complexity. In the query phase, we need to store (1) every data point; (2) the index of every selected t-level vector, and (3) the suffixes of connected neighbors of data points and seed points. Since the data size decreases exponentially from the second layer and the length of the seed point is independent of the dimension, the space for storing seed points is negligible. Therefore, the total space complexity of HVS in the query phase is $O(nd + 2nv_0 + nv_1)$. In contrast, the space complexity of HNSW is around $O((1+\tau)nd)$, where $\tau < 1$ depends on the user-specified parameter and is generally less than 0.01 in practice. Since v_0 and v_1 are bounded

and much smaller than d for high-dimensional datasets, the data size $O(nd)$, dominates the total space complexity in practice.

In the indexing phase, the space complexities of HNSW and HVS are equal to $O(nd + nv_1)$. This is because different HVS layers are constructed independently. Although we need to connect successive layers, this process can be finished before graph construction since the layer-to-layer connection only depends on the selection of seed points and data points.

5.2 Handling updates

Almost all graph-based methods, including NSG and HNSW, could not support deletions and modifications efficiently because all of them are directed graphs. For this reason, we only focus on insertions. Noting that, although NSG and its variant NSSG have single-layer structures, they could not support insertions because they need to be transformed from a k-NN graph, whose topology is affected by any update. Therefore, we only focus on the comparison between HVS and HNSW. In order to make HVS support insertions, we recommend to build three layers consisting of the top layer, the base layer and only a single layer containing quantized vectors ($T = 4$). The algorithm about the insertions in HVS is shown in Algorithm 3. We can see that the whole process can be finished in a single round because the nearest neighbors found in the HVS layer containing seed points are treated as the entering points in the base layer, which does not require an additional search from the scratch. Since the complexity of selecting neighbors from P is constant, insertion and searching have the same time complexity. In addition, since the role of seed points is to partition the data space into lots of Voronoi cells, we recommend not to re-learn seed points unless the change of data space, which rarely occurs in practice.

Clearly, in order to compare the accuracies of insertions of HVS and HNSW, we only need to compare their KNN search performances because the neighbors of every inserted point are chosen from the returned approximate nearest neighbors. From the experimental results in Sec. 6, we will see that, with close or even less indexing time (or running time), HVS could provide more accurate query results than HNSW, which shows that HVS can support insertions better.

6 EXPERIMENTS

6.1 Experimental setup

All benchmarks were implemented in C++. All experiments were performed on a PC with Intel(R) Xeon(R) Gold 6262V CPU@1.90Ghz with 200GB memory, running in Ubuntu 18.04.

6.1.1 Datasets and query sets. We used six real datasets with different sizes and dimensions (see Table 2). These datasets are widely used for the evaluation of various ANNS methods. Since all compared methods are not scalable for the original dataset of Tiny80M on our PC, we represent every data point in Tiny80M with a 150-d vector by the reduction of dimension. For every dataset, we use its original test set, which is independent of the dataset, as the query set. In addition, all reported results are average results over the queries.

Table 2: Data statistics

Dataset	Size	Dim	No. query	Type
Seismic	1,000,000	256	100	Time series
Gist	1,000,000	960	1000	Image
Glove	2,196,017	300	1000	Text
ImageNet	2,340,373	150	200	Image
Tiny5M	5,000,000	384	1000	Image
Tiny80M	79,302,017	150	1000	Image

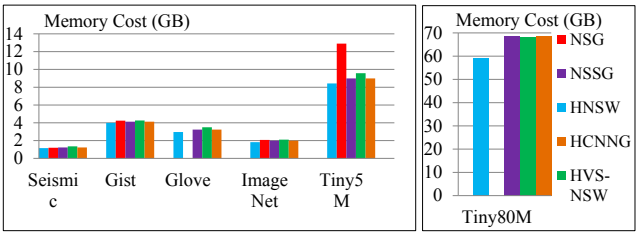
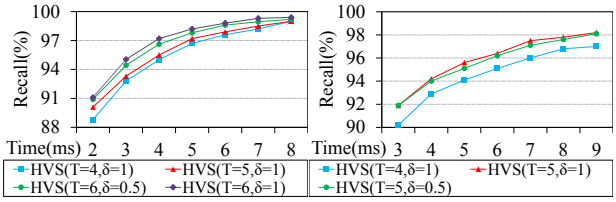


Figure 6: Comparison on the memory cost



(a) Gist, K = 100 **(b) Tiny5M, K = 100**

Figure 7: The efficiency of multiple layers

6.1.2 Benchmark methods and parameter setting. We select four graph-based methods HNSW [14], NSG [30], NSSG [13] and HCNNG [35] as benchmark methods since they have shown their superiority over other in-memory ANNS methods, such as tree-based methods, DPG [25], Faiss (quantization-based methods), FALCONN (LSH-based methods), etc, by extensive experiments [13, 14, 30] (L&C is implemented in Faiss). For K-ANNS problem, we use recall rate, which equals the ratio of the number of successfully returned top-K points to K, to measure the query accuracy.

- **HNSW** [30]. We set parameter *efConstruction* to 1024 which has been regarded as a standard value in the implementations of recent NSW-based methods [26, 37]. The maximum out-degree v in the base layer was set to 32, as suggested in its original code.
- **NSG** [14]. We set its internal parameters L, R and C to 40, 50, 500, respectively. We found NSG performed well on each dataset under such settings.
- **NSSG** [13]. NSSG can be regarded as a variant of NSG. We set its internal parameters L, R and *Angle* to 500, 60, 60, respectively, as suggested in its original code. By such setting, the out-degree of NSSG is bounded by 60.
- **HCNNG** [35]. As suggested by its authors, the number of clusters and the iteration number for K-means were set to 1000 and 20 respectively on all datasets.

- **HVS-NSW.** We combined HVS with the way of linking used by HNSW, and call the new structure as HVS-NSW (For simplicity, we also use the name HVS, which refers to HVS-NSW, in the rest of this paper). For each layer, *efConstruction* was set to 500. Later, we will show that, even with a smaller value of *efConstruction*, the quality of HVS-NSW index is much higher than that of HNSW. In addition, the maximum out-degree v in each layer expect for the top layer was also set to 32. For user-specified parameters T and δ in Algorithm 1, their values were determined based on the data dimension. Specifically, we set T to 4 for datasets with low dimensions (ImageNet, Tiny80M), and disabled the density-based allocation strategy. For datasets with high dimensions (Gist), we set T and δ to 6 and 0.5, respectively. For the other datasets (Word, Glove, Tiny5M), T and δ were set to 5 and 0.5, respectively. The number of training samples was set to 100,000 on Tiny80M and set to 10,000 on other datasets.

For every compared graph-based method, we used parameter $efSearch$ which represents the size of the priority queue, to control the tradeoff between query accuracy and efficiency. When we use a larger value of $efSearch$, we can achieve a higher recall rate at the expense of more running time. In addition, we use K to denote the number of returned points.

It is notable that, although HVS can be combined with NSG or NSSG, we only focus on HVS-NSW in this paper. This is because only HVS-NSW could support insertions, which is deemed to be important in the database field.

6.2 Memory cost

First, we need to point out that the memory cost measured here is different from the index size, especially for NSG/NSSG. In fact, for NSG and NSSG, they need to allocate a space to every point for storing the neighbors of this point before the query phase. The size of such space is determined by the maximum out-degree of NSG or NSSG. This means that their memory costs may be much larger than their index sizes on disk. For this reason, we measure the memory cost rather than the index size because we believe that the former one could reflect the true space cost more accurately. Also, for this reason, the results reported below are not contradictory to the results in the paper of NSG.

The results are shown in Fig. 6 and we have the following observations. (1) Although NSG and NSSG have single layers, they require more space cost than HNSW because of their larger maximum out-degrees. (2) NSG is not scalable on Glove and Tiny80M since its maximum out-degree on these two datasets is above 80,000 and 5,000, respectively. (3) The additional memory cost of HVS-NSW over HNSW arises from the storage of seed points in upper layers. Since the dimensions of seed points are fixed, as the dimension of the original dataset increases, the proportion of such additional cost decreases. An evidence is that, on the high-dimensional dataset Gist, the difference becomes very marginal.

6.3 Indexing time and training time

The results are listed in Table 3 and we have the following observations. (1) Except for Seismic and Gist, the total indexing time of HVS-NSW is less than that of HNSW on other datasets. This is because, for HVS-NSW, the construction time of upper layers is much less than that of the base layer thanks to the efficient distance computations. On the other hand, by means of these upper layers, *efConstruction* of HVS-NSW can be selected to be smaller than that of HNSW to improve the efficiency of indexing. As the data size grows, the construction time of base layer dominates and the advantage of HVS-NSW becomes obvious.

(2) The indexing time of NSG or NSSG is higher than that of HVS-NSW because they need to build k-NN graphs first. Although compared with NSG, NSSG saves much indexing time, the indexing time of NSSG increases sharply as the data size increases (see its results on Tiny80M) while the indexing of HVS-NSW is consistently efficient.

6.4 Search performance

6.4.1 The efficiency of multiple levels. In Sec. 6.4.1, we would like to compare the performances of HVS-NSW under different settings of T, which controls the number of layers. The results on Gist and Tiny5M are shown in Fig. 7. From the results, we have the following observations. (1) Generally, the search performance of HVS-NSW can be improved by increasing the number of layers. This is because finer Voronoi cells could lead to closer entering points in the base layer. (2) We can see that, given the same value of T, the performance of HVS-NSW under $\delta = 0.5$ is fairly close to that under $\delta = 1$, which demonstrates the efficiency of the density-based allocation strategy.

6.4.2 The comparison study. For each dataset, we plotted recall-time curves to compare the performances of four methods by adjusting their values of *efSearch*. Since the difficulties of searching these real datasets are highly different, we chose the target recalls independently for each dataset such that the comparisons under these recalls are informative.

The results are shown in Fig. 8 and Table 4. We have the following observations from different perspectives.

(1) **Running time vs. Recall rate**. From results in Fig. 8, we can see that, in all cases except for $K = 1$ on Tiny80M, HVS-NSW achieved higher recalls than the other methods for each target running time. Even for $K = 1$ on Tiny80M, the difference between HVS-NSW and NSSG is very small. This shows that HVS-NSW is generally superior to the other graph-based methods in the search performance.

(2) **Results for small target times**. From results in Table 4, We can see that, HVS-NSW has the highest recalls among four compared methods on each dataset for the small target running time (1 ms or 2 ms). This shows that HVS-NSW could reach the objective region in which the nearest neighbors fall much faster than the other methods, which supports our motivation to speed up the search in the first phase by using HVS.

(3) **The effect of varying** K. We chose two target recalls 90% and 95%, and compared the performances of four methods under different values of K on Gist and Tiny80M (Gist has the highest dimension and Tiny80M has the largest size). We can see that HVS-NSW incurs less running time than the other methods consistently, which shows the robustness of HVS-NSW in solving the K-ANNS problem.

(4) **Choose HVS or a complicated edge-selection strategy?**. Since NSG and NSSG select edges more carefully than HNSW, their

Table 3: Training time and Indexing time (s); baseline: HNSW. The total times of HVS-NSW and NSG are reported in the form of $A + B$. For HVS-NSW (abbreviated as HVS), A denotes the training time and B denotes the indexing time. For NSG/NSSG, A denotes the construction time of k-NN graph and B denotes the indexing time of NSG/NSSG. In addition, NSG is not scalable on Glove and Tiny80M.

Method	Seismic	Gist	Glove	ImageNet	Tiny5M	Tiny80M
HNSW	**161**	**567**	443	433	1798	20105
NSG	208+4735(30.7X)	703+673(3.1X)	\	304+126(1.0X)	2236+2835(2.8X)	\
NSSG	208+31(1.5X)	703+48(1.7X)	411+4972(12X)	304 + 35(0.9X)	2236+168(1.3X)	22328+47693(3.5X)
HCNNG	434(2.7X)	1514(2.7X)	591(1.3X)	377(0.9X)	2016(1.1X)	17012(0.8X)
HVS	36+157(1.2X)	251+416(1.5X)	**54+333(0.9X)**	**11+232(0.6X)**	**105+1589(0.9X)**	**308+15520(0.8X)**

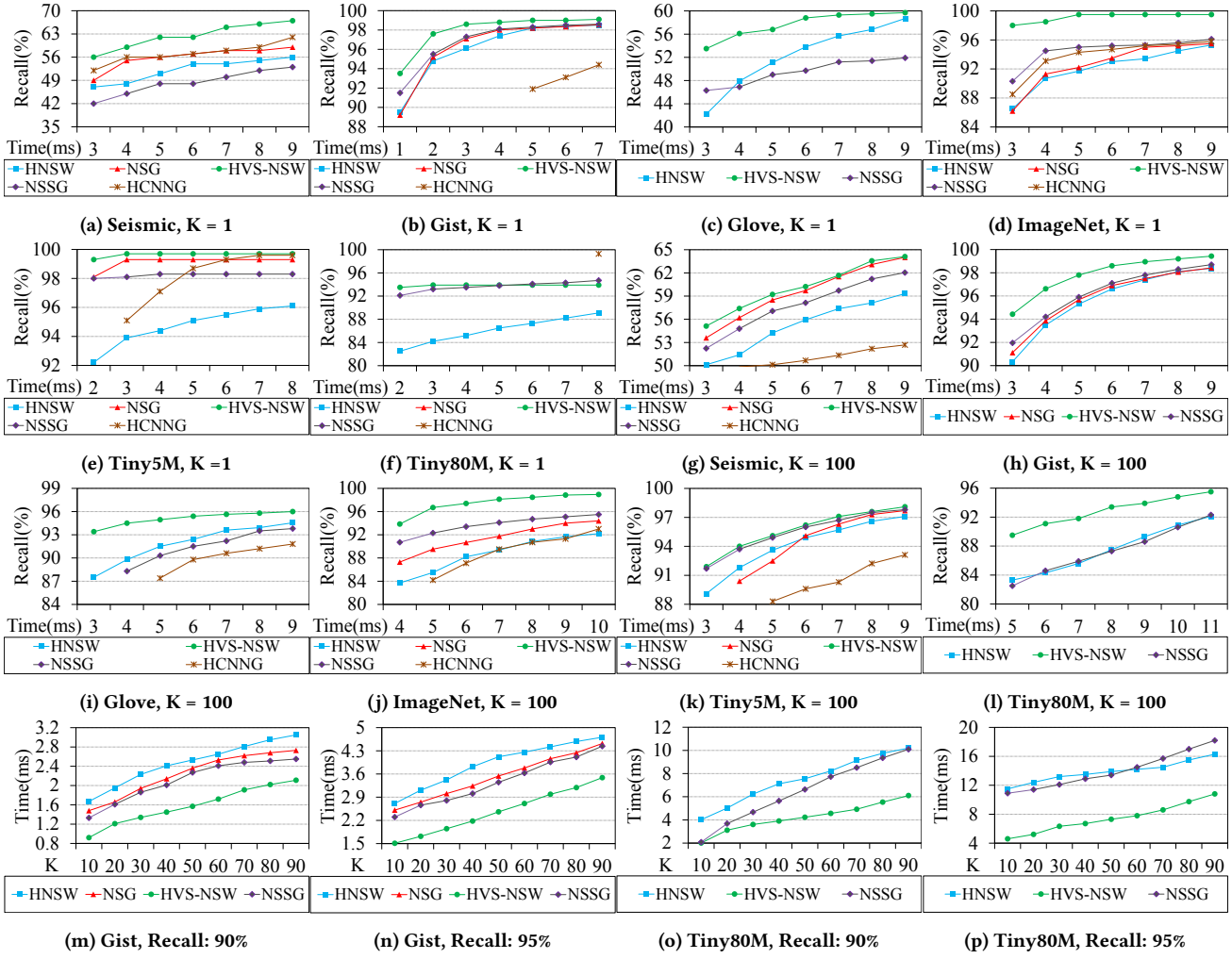


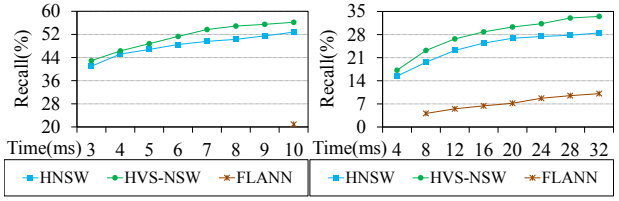
Figure 8: The performances on real datasets. If some method could not achieve target recalls within target running times, we did not report its results.

performances are generally superior to the performance of HNSW. However, an interesting phenomenon is that, HVS-NSW brings more obvious improvement in the search performance over HNSW although both HVS-NSW and HNSW use the same edge-selection strategy. This implies that HVS might be a more efficient approach to improve the performance of existing graph-based methods than just using another sophisticated edge-selection strategy which usually makes the indexing and updates inefficient.

6.4.3 Extension to range search. For HNSW/HVS-NSW, we adopted the following two-phases search strategy to support the range

Table 4: Recalls(%) under small target running times (K = 100 and each target value x (ms) is listed in the form of @x). NSSG/NSG required at least 1.6 ms to return results on Tiny5M and NSSG required at least 5 ms to return results on Tiny80M.

Dataset	HNSW	HVS	NSG	NSSG	HCNNG
Seismic@1	40.4	**44.8**	42.5	42.3	34.7
Gist@2	80.3	**88.5**	85.4	86.2	60.7
Glove@1	76.3	**88.9**	\	72.6	59.1
ImageNet@1	60.5	**81.7**	74.3	76.2	56.8
Tiny5M@1	69.9	**75.2**	\	\	44.8
Tiny80M@2	70.7	**79.1**	\	\	\



(a) squared range: 0.8 **(b) squared range: 1.6**

Figure 9: Results of range search on Tiny80M

search. In the first phase, we normally conduct the KNN search. In the second phase, we check all neighbours of points in the priority queue. If the processed point falls into the target range, it will be added into the priority queue. The second phase terminates when no new point can be added into the priority queue. All points left in the priority queue are final results. We additionally chose FLANN [31], a widely-used tree-based method for range search, as a benchmark and show the results of Tiny80M in Fig. 9. From the results, we can see that HVS-NSW still has the best performance.

7 CONCLUSION

In this paper, we proposed a general framework called HVS to improve the search performance of existing graph-based ANNS methods. HVS builds multi-level virtual Voronoi diagrams on the original proximity graph to accelerate the ANN search. By means of the effective computations of the distances between the query and seed points of Voronoi cells represented by quantized vectors, the nearest neighbors of the query can be approached in a fast and accurate way.

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI (19H04128, 17H06099, 18H04093, 19K11979, 16H01722 and 21H03555).

REFERENCES

[1] Walid G. Aref, Ann Christine Catlin, Jianping Fan, Ahmed K. Elmagarmid, Moustafa A. Hammad, Ihab F. Ilyas, Mirette S. Marzouk, and Xingquan Zhu. 2002. A Video Database Management System for Advancing Video Database Research. In *Multimedia Information Systems*. 8–17.
[2] Sunil Arya and David M. Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *SODA*. 271–280.
[3] Olivier Bachem, Mario Lucic, and Andreas Krause. 2018. Scalable k -Means Clustering via Lightweight Coresets. In *KDD*. 1119–1127.
[4] Dmitry Baranchuk, Dmitry Persiyanov, Anton Sinitsin, and Artem Babenko. 2019. Learning to Route in Similarity Graphs. In *ICML*. 475–484.
[5] Gérard Biau and Luc Devroye. 2015. Lectures on the Nearest Neighbor Method. (2015), 28–38.
[6] Brankica Bratic, Michael E. Houle, Vladimir Kurbalija, Vincent Oria, and Milos Radovanovic. 2018. NN-Descent on High-Dimensional Data. In *WIMS*. 1–8.
[7] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. *SPTAG: A library for fast approximate nearest neighbor search.* https://github.com/Microsoft/SPTAG
[8] D. Dearholt, N. Gonzales, and G. Kurup. 1988. Monotonic search networks for computer vision databases. *Signals, Systems and Computers* 2 (1988), 548–553.
[9] Wei Dong, Moses Charikar, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*. 577–586.
[10] Matthijs Douze, Alexandre Sablayrolles, and Hervé Jégou. 2018. Fast indexing with graphs and compact regression codes. In *CVPR*. 3646–3654.
[11] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2019. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *PVLDB* 13, 3 (2019), 403–420.
[12] Cong Fu and Deng Cai. 2016. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. In *arXiv:1609.07228*.
[13] Cong Fu, Changxu Wang, and Deng Cai. 2021. High Dimensional Similarity Search with Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Trans. Pattern Anal. Mach. Intell* (2021).
[14] 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.
[15] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Trans. Pattern Anal. Mach. Intell* 36, 4 (2014), 744–755.
[16] Rachid Guerraoui, Anne-Marie Kermarrec, Olivier Ruas, and François Taïani. 2020. Smaller, Faster & Lighter KNN Graph Constructions. In *WWW*. 1060–1070.
[17] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. 2011. Fast Approximate Nearest-Neighbor Search with k-Nearest Neighbor Graph. In *IJCAI*. 1312–1317.
[18] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *CVPR*. 5713–5722.
[19] Qiang Huang, Jianlin Feng, Qiong Fang, Wilfred Ng, and Wei Wang. 2017. Query-aware locality-sensitive hashing scheme for l_p norm. *VLDB J.* 26, 5 (2017), 683–708.
[20] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell* 33, 1 (2011), 117–128.
[21] Zhongming Jin, Debing Zhang, Yao Hu, Shiding Lin, Deng Cai, and Xiaofei He. 2014. Fast and Accurate Hashing Via Iterative Nearest Neighbors Expansion. *IEEE Trans. Cybern* 44, 11 (2014), 2016–2177.
[22] Yan Ke, Rahul Sukthankar, and Larry Huston. 2004. An efficient parts-based near-duplicate and sub-image retrieval system. In *ACM Multimedia*. 869–876.
[23] Yifan Lei, Qiang Huang, Mohan S. Kankanhalli, and Anthony K. H. Tung. 2020. Locality-Sensitive Hashing Scheme based on Longest Circular Co-Substring. In *SIGMOD*. 2589–2599.
[24] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. 2020. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *SIGMOD*. 2539–2554.
[25] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *IEEE Trans. Knowl. Data Eng* 32, 8 (2020), 1475–1488.
[26] Jie Liu, Xiao Yan, Xinyan Dai, Zhirong Li, James Cheng, and Ming-Chang Yang. 2020. Understanding and Improving Proximity Graph based Maximum Inner Product Search. In *AAAI*. 139–146.
[27] Kejing Lu and Mineichi Kudo. 2020. R2LSH: A Nearest Neighbor Search Scheme Based on Two-dimensional Projected Spaces. In *ICDE*. 1045–1056.
[28] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. 2020. VHP: Approximate Nearest Neighbor Search via Virtual Hypersphere Partitioning. In *PVLDB*. 1443–1455.
[29] 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.
[30] Yury A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell* 42, 4 (2020), 824–836.
[31] Marius Muja. 2019. https://github.com/flann-lib/flann.
[32] Liudmila Prokhorenkova and Aleksandr Shekhovtsov. 2020. Graph-based Nearest Neighbor Search: From Practice to Theory. In *ICML*. 7803–7813.
[33] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. 2019. Spreading vectors for similarity search. In *ICLR*.
[34] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: Solving c-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *PVLDB* 8, 1 (2014), 1–12.

[35] Javier Vargas Muñoz, Marcos A. Gonçalves, Zanoni Dias, and Ricardo da S. Torres. 2019. Hierarchical Clustering-Based Graphs for Large Scale Approximate Nearest Neighbor Search. *Pattern Recognition* 96 (2019).

[36] Minjia Zhang and Yuxiong He. 2019. GRIP: Multi-Store Capacity-Optimized High-Performance Nearest Neighbor Search for Vector Search Engine. In *CIKM*. 1673–1682.

[37] Zhixin Zhou, Shulong Tan, Zhaozhuo Xu, and Ping Li. 2019. Mobius Transformation for Fast Inner Product Search on Graph. In *NeurIPS*. 8216–8227.