Three Algorithms for Merging Hierarchical Navigable Small World Graphs

Alexander Ponomarenko[1,2]

[1]Laboratory LATNA, HSE University
[2]Yugabyte, Inc

Abstract

This paper addresses the challenge of merging hierarchical navigable small world (HNSW) graphs, a critical operation for distributed systems, incremental indexing, and database compaction. We propose three algorithms for this task: Naive Graph Merge (NGM), Intra Graph Traversal Merge (IGTM), and Cross Graph Traversal Merge (CGTM). These algorithms differ in their approach to vertex selection and candidate collection during the merge process. We conceptualize graph merging as an iterative process with four key steps: processing vertex selection, candidate collection, neighborhood construction, and information propagation. Our experimental evaluation on the SIFT1M dataset demonstrates that IGTM and CGTM significantly reduce computational costs compared to naive approaches, requiring up to 70% fewer distance computations while maintaining comparable search accuracy. Surprisingly, IGTM outperforms CGTM in efficiency, contrary to our initial expectations. The proposed algorithms enable efficient consolidation of separately constructed indices, supporting critical operations in modern vector databases and retrieval systems that rely on HNSW for similarity search.

Keywords: approximate nearest neighbor search, hierarchical navigable small world, graph merging, vector databases, information retrieval

1 Introduction

The k-nearest neighbor (k-NN) search problem has emerged as a fundamental computational challenge with critical applications across numerous domains. In today's data-driven landscape, efficient similarity search is essential for recommendation systems, computer vision, multimedia retrieval, and natural language processing applications. Recent advancements in generative AI, particularly Retrieval-Augmented Generation (RAG) systems, have further highlighted the importance of fast and accurate nearest neighbor search. RAG frameworks depend on efficient retrieval of relevant document fragments from large vector databases to ground language model outputs in factual information, making high-performance approximate nearest neighbor (ANN) search algorithms increasingly vital.

1.1 Nearest Neighbor Search Problem

Formally, the k-nearest neighbor search problem can be defined as follows: Given a set of points

$$P = \{p_1, p_2, \ldots, p_n\}$$

in a metric space $\big(X, \rho\big)$, where ρ is a distance function, and a query point $q \in X$, find a set

$$P_k \;=\; \underset{S \subseteq P,\ |S|=k}{\arg\min} \sum_{p \in S} \rho(q, p).$$

In other words, we seek the set of k points whose total distance to q is minimal.

Three main approaches have been developed for tackling the nearest neighbor search problem:

1. **Space-partitioning trees**: These methods divide the vector space into hierarchical regions, enabling logarithmic search complexity in low dimensions. Notable examples include KD-trees [5], R-trees [14], VP-trees [30], M-trees [8], and Cover trees [6]. A comprehensive overview of these approaches for metric spaces is provided by Zezula et al. [31].

2. **Mapping-based methods**: These techniques transform the original data into representations that allow for efficient search. This category includes Locality-Sensitive Hashing (LSH) [2, 3, 9, 13, 15] and Product Quantization approaches [12, 17, 22], which encode vectors into compact codes that approximate the distances between points.

3. **Navigable graph-based methods**: The most recent and currently state-of-the-art approach that constructs graph structures where vertices represent data points and edges connect similar points. These graphs are designed to allow "greedy-like" algorithms to perform a directed traversal through the data space.

1.2 Hierarchical Navigable Small World (HNSW)

Hierarchical Navigable Small World (HNSW) [18] is a state of the art navigable graph-based method. It demonstrates superior performance on modern ANN benchmarks [4]. Due to relative simplicity, and efficiency, the HNSW algorithm has become widely used. It has been implemented in popular libraries for ANN-Search [10], [7], and it is implemented as vector index in many database systems including PostgreSQL (vector extension), ORACLE [23], and other vector search oriented databases such as Milvus [21], Zilliz [33], Weaviate [28].

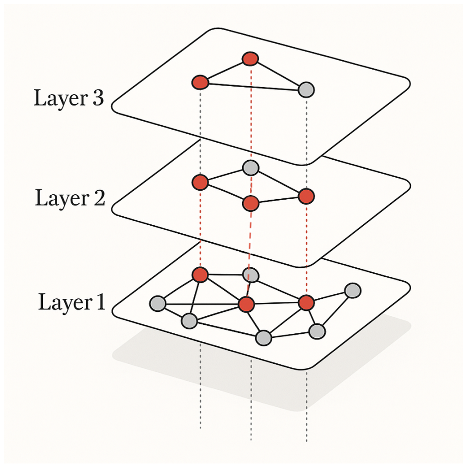

Figure 1: Schematic example HNSW structure. Each layer is a navigable graph that suit for "greedy-like" traversal algorithms. Graphs of all levels glued together form a navigable small world graph similar to NSW. An explicit separation set of edges of different lengths to layers helps to reduce the number of computations in search while traversing vertexes with high degrees, also called hubs.

HNSW [18] extends the navigable graphs concept by organizing points in a hierarchy of navigable graphs also named layers. Each layer contains a progressively smaller subset of data points. The bottommost layer contains all points, while the upper layers form a sparse representation of the data space. Graphs of all levels glued together form a navigable small world graph similar to NSW [19, 20, 24]. An explicit separation set of edges of different lengths to layers helps to reduce the number of computations in search while traversing vertexes with high degrees, also called hubs. A schematic example of HNSW structure presented in Fig. 1.

1.3 Graph Merging Challenge

While HNSW supports efficient dynamic insertions and deletions, an important operational challenge arises in practical systems: merging multiple independently constructed graph indices. This problem is particularly relevant in distributed systems, incremental indexing scenarios, and database compaction operations. In graph-based indices, deletion operations are often implemented by simply marking vertices as deleted without actually removing them, to avoid disrupting the graph's connectivity. This approach prevents the computationally expensive task of rebuilding neighborhoods for affected vertices. However, as deletions accumulate, the index becomes increasingly sparse and less efficient.

Graph merging provides a way to periodically compact the structure by creating a fresh graph that excludes deleted vertices while preserving the navigability properties essential for efficient search. Additionally, in distributed computing environments, merging separately constructed indices becomes necessary for building unified search structures over partitioned datasets.

1.4 Graph Merging as an Iterative Process

The merge operation can be conceptualized as an iterative process consisting of four key steps:

1. **Processing Vertex Selection**: Choosing a vertex v^* for which we aim to construct a neighborhood in the merged graph and determining which original graph it belongs to.

2. **Candidate Collection**: Gathering potential neighbors from both input graphs. This involves searching within each graph structure to find vertices that should be considered for inclusion in v^*'s neighborhood.

3. **Neighborhood Construction**: Applying a specified strategy (e.g., k-nearest neighbors or relative neighborhood graph) to select the final set of neighbors for v^* from the candidates.

4. **Information Propagation**: Deciding what information from the current iteration should be preserved for the next iteration to optimize the process, then proceeding to step 1 with a new vertex.

By carefully designing each of these steps, we can create merge algorithms that balance computational efficiency with the quality of the resulting graph structure.

In this work, we propose three novel algorithms based on this iterative framework—**Naive Graph Merge (NGM)**, **Intra Graph Traversal Merge (IGTM)**, and **Cross Graph Traversal Merge (CGTM)**—for efficiently merging HNSW graphs. These algorithms differ primarily in how they implement each step of the iterative process. For instance, NGM and IGTM select the next vertex for neighborhood construction only from the same graph as the current vertex. In contrast, CGTM implements the idea that the next vertex for neighborhood construction can be selected from any input graph.

The rest of this paper is organized as follows. Section 2 describes the search algorithms for HNSW graphs that we use in merge procedures. Section 3 recalls neighborhood construction strategies that are essential for both graph building and merging. Section 4 introduces our three graph merging algorithms in detail. Section 5 presents computational experimental results. We give references to similar works in Section 6. Finally, Section 7 concludes the paper and discusses directions for future research.

2 Search

The search process in navigable graphs is a critical operation that underlies both query processing and the graph construction phase. Below, we describe two key search algorithms: LocalSearch for exploring a single graph layer, and HNSW-Search for hierarchical multi-layer traversal.

2.1 Local Search

Algorithm 1 LocalSearch(G, q, C, k, L)

Input: Graph $G = (V, E)$, query $q \in \mathbb{R}^d$, initial candidate set $C \subset V$, $k \in \mathbb{N}$, $L \in \mathbb{N}$

Output: Approximate k-nearest neighbors $V^* \subset V$

```
1: while True do
2:     $u \leftarrow$ nearest unvisited point to $q$ in $C$
3:     $U \leftarrow \{v \mid (u, v) \in E\}$
4:     for $v \in U$ do
5:         if $v$ is unvisited then
6:             $C \leftarrow C \cup \{v\}$
7:     if $|C| > L$ then
8:         $C \leftarrow$ top $L$ nearest points to $q$ in $C$
9:     if no updates to $C$ then
10:        break
11: return top-$k$ nearest points to $q$ in $C$
```

The LocalSearch algorithm (Algorithm 1) implements a “greedy” like search within a single graph layer. Starting with an initial candidate set C, it iteratively explores the neighborhood of the closest unvisited point to the query q. This exploration strategy balances depth-first search for quick convergence toward the target region with breadth-first search for escaping local minima. In the literature, this algorithm is also named ”beam search” [25], [29]. It was introduced in [24] paper, and later adopted in HNSW in other graph-based methods as a basic search algorithm [16], [27].

For each iteration, LocalSearch selects the closest unvisited point u to the query and explores its immediate neighbors. These neighbors are added to the candidate set if they have not been visited previously. The candidate set is constrained to size L (the expansion factor) by retaining only the L points closest to the query. This pruning step is crucial for an efficient search by focusing computational resources on the most promising candidates.

The search terminates when no additional updates to the candidate set occur during an iteration, indicating convergence. The algorithm then returns the k nearest points to the query from the final candidate set, which represents the approximate k-nearest neighbors of the query within the graph structure.

The parameter L provides a direct trade-off between search quality and computational cost. Larger values of L allow for a broader exploration, potentially escaping local minima and finding better global solutions, but at the expense of increased computation time. In the HNSW paper, the parameter L is named as **ef** (expansion factor) parameter.

2.2 Hierarchical Search

Algorithm 2 HNSW-Search($\mathcal{H}, q, v_0, k, L, \ell$)

Input: HNSW graph $\mathcal{H} = (G_i)_{i=0}^{l_{\max}}$, query $q \in \mathbb{R}^d$, starting vertex $v_0 \in V$, $k, L \in \mathbb{N}$, search layer ℓ
Output: Approximate k-nearest neighbors $V^* \subset V$

```
1: $v^* \leftarrow v_0$
2: for $i = l_{\max}$ down to $\ell$ do
3:     $v^* \leftarrow$ LocalSearch($G = G_i, q = q, C = \{v^*\}, k = 1, L = L$)
4: return LocalSearch($G_\ell, q, \{v^*\}, k, L$)
```

The HNSW-Search algorithm (Algorithm 2) leverages the hierarchical structure of HNSW to efficiently navigate through the vector space. The search begins at the highest layer $l_{\max}$ of the HNSW structure with a single entry point v_0. At each layer, LocalSearch is used to find the best approximation of the nearest neighbor to the query within that layer.

The algorithm traverses from the highest layer down to the target layer ℓ, using the result from each layer as the entry point for the next lower layer. This coarse-to-fine approach allows the search to quickly focus on the relevant region of the vector space at higher, sparser layers before refining the search in the more densely connected lower layers.

At each layer i (from $l_{\max}$ down to $\ell + 1$), LocalSearch is executed with $k = 1$ to find a single nearest neighbor to the query. This single neighbor serves as an entry point for the subsequent layer. At the target layer ℓ, a final LocalSearch is performed with the specified value k to retrieve the k-nearest neighbors.

This hierarchical search significantly reduces the number of distance computations compared to a flat graph search, especially for large-scale datasets. The efficiency comes from the progressive narrowing of the search space as the algorithm moves down through the layers, focusing the search on increasingly relevant regions.

Both search algorithms are fundamental not only for answering queries but also for the merge operations described in subsequent sections, as they provide the mechanism for identifying candidate neighbors when reconstructing connections in the merged graph.

HNSW-Search algorithm is a default search algorithm that is exposed to the user in all popular HNSW implementations with a default value of the parameter $\ell = 0$, and assuming that v_0 is one of the vertex of the top layer [26], [1], [10].

In the text, we also refer to the HNSW-Search algorithm as a "standard", or "default search" algorithm. The LocalSearch function is hidden from the user and is used only as a part of a particular implementation".

3 Neighborhood Construction Strategies

Algorithm 3 KNN-Neighborhood-Construction(v^*, C, k)

```
Input: Vertex $v^*$, candidate set $C$, number of neighbors $k$
Output: Set $C'$ of $k$ closest neighbors
1: $C' \leftarrow k$-nearest neighbors of $v^*$ in $C$
2: return $C'$
```

Various strategies exist for selecting a node's neighborhood to form a navigable property for a graph. Our merging algorithms are designed to be agnostic to the specific neighborhood construction method, which can be provided as a parameter. We present two representative strategies below.

The KNN-Neighborhood-Construction algorithm (Algorithm 3) implements the simplest approach, where a vertex's neighborhood consists of its k-nearest neighbors from the candidate set. This strategy prioritizes proximity, but may not optimize for graph navigability properties.

In contrast, the RNG-Neighborhood-Construction algorithm (Algorithm 4) implements a more sophisticated approach based on relative neighborhood graph principles. It processes candidates in ascending order of distance to the target vertex v^*. For each candidate v, it checks whether v is closer to v^* than to any previously selected neighbor w. This pruning condition helps create better-connected graphs with improved navigability by ensuring that edges span different directions in the vector space rather than clustering in the same region. The algorithm limits the neighborhood size to at most m vertices to control maximum vertex degree.

The choice of neighborhood construction strategy significantly impacts both search performance and the computational cost of index maintenance operations, including merges. While KNN-Neighborhood-Construction is computationally simpler, RNG-Neighborhood-Construction typically produces graphs with better search performance at the expense of more complex neighborhood formation. In this paper, we performed our experiments with RNG-Neighborhood-Construction.

4 Merge Algorithms

We now present three algorithms for merging HNSW graphs: NGM, IGTM, and CGTM. These algorithms operate layer by layer. We first describe the general procedure for merging the multi-layer HNSW structure, followed by the specific logic for merging a single layer (*layer-merge algorithms*).

4.1 Simple Insertion Graph Merge

Before we start, we need to provide a short description of simple insertion graph merge (SIGM). It is a default merging strategy that uses an insertion operation

Algorithm 4 RNG-Neighborhood-Construction(v^*, C, m)

Input: Vertex v^*, candidate set C, maximum neighborhood size m
Output: Filtered neighbor set C'

```
1: Sort $C$ in ascending order by distance to $v^*$
2: $C' \leftarrow \emptyset$
3: for $v \in C$ do
4:     $f \leftarrow$ true
5:     for $w \in C'$ do
6:         if $\rho(v^*, v) \geq \rho(v, w)$ then
7:             $f \leftarrow$ false
8:             break
9:     if $f$ then
10:        $C' \leftarrow C' \cup \{v\}$
11:    if $|C'| \geq m$ then
12:        break
13: return $C'$
```

to merge graphs. It takes the largest graph and sequentially inserts data from other graph. Actually, it is not a merging procedure, but we use it as a basic benchmark.

4.2 HNSW General Merge Framework

The HNSW-General-Merge algorithm (Algorithm 5) provides a framework for merging two HNSW structures, $\mathcal{H}_a$ and $\mathcal{H}_b$. It iterates through each layer level and applies a chosen layer-merge algorithm (NGM, IGTM, or CGTM) to combine the corresponding layers G_i^a and G_i^b. The layer-merge algorithms require access to the full HNSW structures $\mathcal{H}_a, \mathcal{H}_b$ to perform searches using HNSW-Search.

Algorithm 5 HNSW-General-Merge($\mathcal{H}_a, \mathcal{H}_b$, LayerMergeAlgo, params)

Input: HNSW graphs $\mathcal{H}_a$, $\mathcal{H}_b$. Chosen layer merge algorithm LayerMergeAlgo (e.g., NGM). Algorithm-specific parameters params.
Output: Merged HNSW graph $\mathcal{H}_c = (G_i^c)_{i=0}^{l_{\max}^c}$

```
1: $l_{\max}^a \leftarrow \mathcal{H}_a$.getMaxLayerNumber()
2: $l_{\max}^b \leftarrow \mathcal{H}_b$.getMaxLayerNumber()
3: $l_{\max}^c \leftarrow \max(l_{\max}^a, l_{\max}^b)$
4: for $i = 0$ to $l_{\max}^c$ do                                    ▷ Merge each layer
5:     $G_i^c \leftarrow$ LayerMergeAlgo($\mathcal{H}_a, \mathcal{H}_b, \ell = i$, params)  ▷ Calls e.g., Merge-Naive
6: $\mathcal{H}_c \leftarrow (G_i^c)_{i=0}^{l_{\max}^c}$
7: return $\mathcal{H}_c$
```

4.3 Naive Graph Merge

The Naive Graph Merge (NGM) algorithm (Algorithm 6) provides a straightforward method for merging a single layer ℓ. The algorithm begins by extracting the target layers from both input HNSW structures and initializing the merged graph's vertex set as the union of vertices from both input graphs (lines 1-4).

For each vertex v^* in graph G^a_ℓ (line 5), the algorithm:

1. Searches for potential neighbors in graph G^b_ℓ using HNSW-Search (line 6)
2. Combines these candidates with v^*'s original neighbors from G^a_ℓ (line 7)
3. Selects the final neighborhood for v^* using the specified Neighborhood-Construction strategy and adds the corresponding edges to E^c (line 8)

The same process is then repeated for all vertices of graph G^b_ℓ (lines 9-12).

This approach ensures that each vertex in the merged graph has an appropriate neighborhood that incorporates information from both input graphs. However, it is computationally intensive due to repeated HNSW-Search calls that traverse multiple layers for each vertex.

Algorithm 6 NGM($\mathcal{H}_a, \mathcal{H}_b, \ell$, NeighborhoodConstruction, m, search_ef, v_{entry})

Input: HNSW graphs $\mathcal{H}_a$, $\mathcal{H}_b$; target layer ℓ; Neighborhood construction function NeighborhoodConstruction; target neighborhood size m; search parameter search_ef; entry point v_{entry} (e.g., from $\mathcal{H}_a$ or $\mathcal{H}_b$)
Output: Merged graph G^c

```
1: $G^a \leftarrow \mathcal{H}_a$.GetLayer($\ell$); $G^b \leftarrow \mathcal{H}_a$.GetLayer($\ell$)
2: $V^a \leftarrow$ vertices($G^a$); $V^b \leftarrow$ vertices($G^b$)
3: $E^a \leftarrow$ edges($G^a$); $E^b \leftarrow$ edges($G^b$)
4: $V^c \leftarrow V^a \cup V^b$; $E^c \leftarrow \emptyset$
5: for $v^* \in V^a$ do
6:     $\mathcal{C}^b \leftarrow$ HNSW-Search($\mathcal{H} = \mathcal{H}_b, q = v^*, v_{entry} = v^b_{entry}, k = m, L =$ search_ef$, \ell = \ell$)
7:     $\mathcal{C} \leftarrow \{v \mid (v^*, v) \in E^a\} \cup \mathcal{C}^b$
8:     $E^c \leftarrow E^c \cup \{(v^*, v) | v \in$ neighborhood_construction$(\mathcal{C}, v^*, m)\}$
9: for $v^* \in V^b$ do
10:     $\mathcal{C}^a \leftarrow$ HNSW-Search($\mathcal{H} = \mathcal{H}_a, q = v^*, v_{entry} = v^a_{entry}, k = m, L =$ search_ef$, \ell_{target} = \ell$)
11:     $\mathcal{C} \leftarrow \{v \mid (v^*, v) \in E^b\} \cup \mathcal{C}^a$
12:     $E^c \leftarrow E^c \cup \{(v^*, v) | v \in$ neighborhood_construction$(\mathcal{C}, v^*, m)\}$
13: return $G^c = (V^c, E^c)$
```

Algorithm 7 IGTM($\mathcal{H}_a, \mathcal{H}_b, \ell, \text{jump\_ef}, \text{local\_ef}, \text{next\_step\_k}, M, m$)

Input: The HNSW graphs $\mathcal{H}_a = (G_i^a), \mathcal{H}_b = (G_i^b)$, the merging layer number ℓ, the size of the forming neighborhoods $m \in \mathbb{N}$, parameters $\text{jump\_ef}, \text{local\_ef}, \text{next\_step\_k} \in \mathbb{N}$

Output: Merged graph G^c

```
1: $G^a \leftarrow \mathcal{H}_a$.GetLayer($\ell$); $G^b \leftarrow \mathcal{H}_a$.GetLayer($\ell$)
2: $V^a \leftarrow$ vertices($G^a$); $V^b \leftarrow$ vertices($G^b$)
3: $E^a \leftarrow$ edges($G^a$); $E^b \leftarrow$ edges($G^b$)
4: $V^c \leftarrow V^a \cup V^b$; $E^c \leftarrow \emptyset$
5: $\mathcal{V}_{not\_done} \leftarrow V^a$
6: while $\mathcal{V}_{not\_done} \neq \emptyset$ do
7:     $v^* \leftarrow$ random choice from $\mathcal{V}_{not\_done}$
8:     $\mathcal{P}^b \leftarrow$ HNSW-Search($\mathcal{H} = \mathcal{H}^b, q = v^*, v_0, k = M, L = \text{jump\_ef}, \ell$)
9:     while True do
10:        $\mathcal{V}_{not\_done} \leftarrow \mathcal{V}_{not\_done} \setminus \{v^*\}$
11:        $\mathcal{C}^b \leftarrow$ LocalSearch($G = G^b, q = v^*, C = \mathcal{P}^b, k = m, L = \text{local\_ef}$)
12:        $\mathcal{C} \leftarrow \{v : (v^*, v) \in E^a\} \cup \mathcal{C}^b$
13:        $E^c \leftarrow E^c \cup \{(v^*, v) : v \in$ NeighborhoodConstruction($\mathcal{C}, v^*, m$)$\}$
14:        $\mathcal{P}^b \leftarrow \{\mathcal{C}_1^b, \mathcal{C}_2^b, ..., \mathcal{C}_M^b\}$
15:        $\mathcal{C}^a \leftarrow$ LocalSearch($G = G^a, q = v^*, C = \{v^*\}, k = \text{next\_step\_k}, L = \text{next\_step\_ef}$)
16:        $\mathcal{C}^a \leftarrow \mathcal{C}^a \cap \mathcal{V}_{not\_done}$
17:        if $\mathcal{C}^a = \emptyset$ then
18:            break
19:        $v^* \leftarrow \mathcal{C}_1^a$
20: $\mathcal{V}_{not\_done} \leftarrow V^b$
21: while $\mathcal{V}_{not\_done} \neq \emptyset$ do
22:     Repeat the same process for $V^b$ with the roles of $\mathcal{H}_a$ and $\mathcal{H}_b$ swapped.
23: return $G^c = (V^c, E^c)$
```

4.4 Intra Graph Traversal Merge

The most effort of the NGM algorithm lies in obtaining the set of neighborhood candidates from the other graph utilizing the HNSW-Search procedure, which every time traverses the layer graphs from the top level down to the layer number ℓ. The number of computations can be reduced if we select the next vertex to process v^* close to the previous one (line 15 of the algorithm 7), instead of randomly choosing it. Thus, for the new v^* the neighborhood candidates will also be close to the previous candidates set. To search for these new neighborhood candidates we can use the LocalSearch procedure which traverses the same graph staring from the previous neighborhood candidates set $\mathcal{P}^b$ (lines 11). In line 14 in the set $\mathcal{P}^b$ we keep only M-closest to v^* candidates.
In lines 15,16, we select a new processing vertex v^* close to the previous processing vertex v^* that was not already processed. To ensure that the new processing vertex v^* is not very far from the previous v^* we bound the size of the results of the LocalSearch procedure controlling by next_step_k parameter. Once LocalSearch cannot find enough close unprocessed vertex (if condition in line 17), in line 7 we choose a new v^* from the set $\mathcal{V}_{not\_done}$ randomly.
After we have processed all vertices from the graph G^a. We do the same for the vertices of graph G^b (lines 21-22).

4.5 Cross Graph Traversal Merge

Cross Graph Traversal Merge (CGTM) algorithm is similar to IGTM utilizes the LocalSearch procedure to reduce computation efforts. The difference is that the IGTM algorithm chooses the next processing vertex v^* from the same graph, while CGTM looks for the new processing vertex v^* in both graphs G^a, and G^b. Thus in line 24 v^* is chosen from the set $\mathcal{C}_{not\_done}$, which is one of the not processed vertex of both graph (line 21). The intuition underlying CGTM algorithm is that when we choose a new processing vertex v^* from both graphs, we reduce the number of times when v^* is chosen randomly, thus minimizing the number of times that we use the more expensive search procedure HNSW-Search.

5 Computational Experiments

5.1 Setup

We evaluated the proposed merge algorithms on the standard ANN benchmark dataset SIFT1M of 1 million 128-dimensional vectors. We divided it into two disjoint subsets of 500k vectors. In these subsets, we built two HNSW indices $\mathcal{H}_a$ and $\mathcal{H}_b$ using the following parameters: $M = 16$, $M0 = 32$, ef_construction $= 32$. Then these indices were merged using a simple insertion strategy SIGM and the three proposed algorithms: NGM, IGTM, and CGTM. We keep the same values of the parameters $m = 16$ and $m0 = 32$ as the target neighborhood size in the merged graph.

Algorithm 8 CGTM($\mathcal{H}_a, \mathcal{H}_b, \ell, \text{jump\_ef}, \text{local\_ef}, \text{next\_step\_k}, M, m$)

Input: The HNSW graphs $\mathcal{H}_a = (G_i^a)$, $\mathcal{H}_b = (G_i^b)$, the merging layer ℓ, parameters jump_ef, local_ef, next_step_k $\in \mathbb{N}$, neighborhood sizes $M, m \in \mathbb{N}$

Output: Merged graph G^c

```
1: $G^a \leftarrow \mathcal{H}_a$.GetLayer($\ell$); $G^b \leftarrow \mathcal{H}_a$.GetLayer($\ell$)
2: $V^a \leftarrow$ vertices($G^a$); $V^b \leftarrow$ vertices($G^b$)
3: $E^a \leftarrow$ edges($G^a$); $E^b \leftarrow$ edges($G^b$)
4: $V^c \leftarrow V^a \cup V^b$; $E^c \leftarrow \emptyset$
5: $\mathcal{V}_{not\_done} \leftarrow V^a \cup V^b$
6: while $\mathcal{V}_{not\_done} \neq \emptyset$ do
7:     $v^* \leftarrow$ randomly choice from $\mathcal{V}_{not\_done}$
8:     $\mathcal{P}^a \leftarrow$ HNSW-Search($\mathcal{H} = \mathcal{H}^a, q = v^*, v_0, k, L = \text{jump\_ef}, \ell$)
9:     $\mathcal{P}^b \leftarrow$ HNSW-Search($\mathcal{H} = \mathcal{H}^b, q = v^*, v_0, k, L = \text{jump\_ef}, \ell$)
10:    while True do
11:        $\mathcal{V}_{not\_done} \leftarrow \mathcal{V}_{not\_done} \setminus \{v^*\}$
12:        $\mathcal{C}^a \leftarrow$ LocalSearch($G = G^a, q = v^*, C = \mathcal{P}^a, k = m, L = \text{local\_ef}$)
13:        $\mathcal{C}^b \leftarrow$ LocalSearch($G = G^b, q = v^*, C = \mathcal{P}^b, k = m, L = \text{local\_ef}$)
14:        if $v^* \in V^a$ then
15:            $\mathcal{C} \leftarrow \{v : (v^*, v) \in E^a\} \cup \mathcal{C}^b\}$
16:        else
17:            $\mathcal{C} \leftarrow \{v : (v^*, v) \in E^b\} \cup \mathcal{C}^a\}$
18:        $E^c \leftarrow E^c \cup \{(v^*, v) : v \in \text{neighborhood\_construction}(\mathcal{C}, v^*, m)\}$
19:        $\mathcal{C}^a_{not\_done} \leftarrow \{\mathcal{C}^a_1, \mathcal{C}^a_2, ..., \mathcal{C}^a_{\text{next\_step\_k}}\} \cap \mathcal{V}_{not\_done}$
20:        $\mathcal{C}^b_{not\_done} \leftarrow \{\mathcal{C}^b_1, \mathcal{C}^b_2, ..., \mathcal{C}^b_{\text{next\_step\_k}}\} \cap \mathcal{V}_{not\_done}$
21:        $\mathcal{C}_{not\_done} \leftarrow \mathcal{C}^a_{not\_done} \cup \mathcal{C}^b_{not\_done}$
22:        if $\mathcal{C}_{not\_done} = \emptyset$ then
23:            break
24:        $v^* \leftarrow \underset{v \in \mathcal{C}_{not\_done}}{\text{argmin}} \ \rho(v, v^*)$
25:        $\mathcal{P}_a \leftarrow \mathcal{C}^a$
26:        $\mathcal{P}_b \leftarrow \mathcal{C}^b$
27: return $G^c = (V^c, E^c)$
```

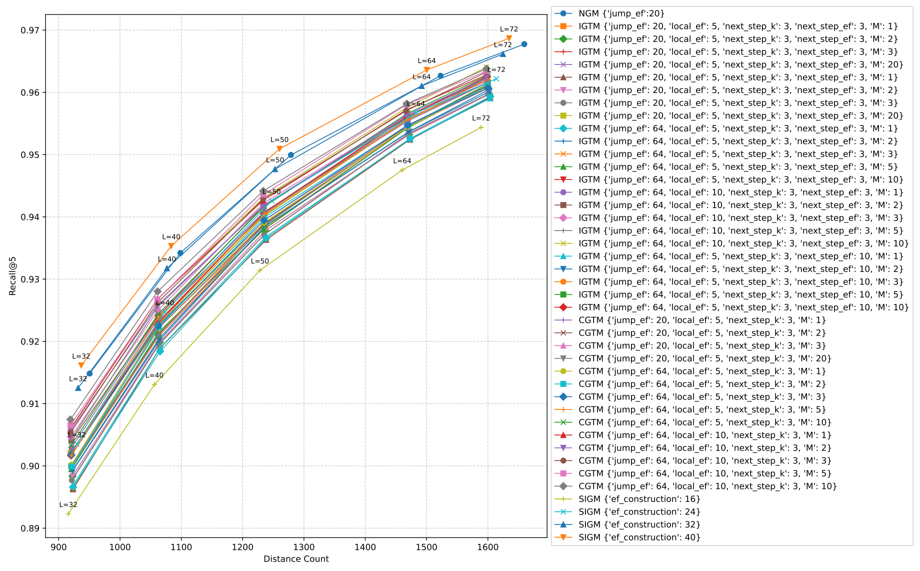

Figure 2: Recall vs distance count on search stage for merged graphs.

In order to be independent of implementation details and low-level optimisations, we made a comparison of the algorithms based on the number of distance computations required for the merge process.

Other important characteristics of the merge process is how accurate the merge is done. In this case, the word "merging accuracy" can mean different things. For example, we can estimate merging accuracy as how similar the neighborhoods of the merged graph to the neighborhoods of the graph constructed by simple insertion strategy SIGM. Another way, which we prefer, is to verify how the merged graph is good for search. To quantify search quality, we performed a standard search performance test on the merged graphs. So, for the merged graphs, we have measured a trade-off between recall and the number of distance computations, by running algorithm 2 with different values of the parameter L (search expansion factor). We measured the recall@5 metric for L = 32, 40, 50, 64, and 72. This metric indicates how often the true nearest neighbors appear in the top-5 results returned by the search algorithm.

Formally, for a given query point q, let $P_k(q)$ denote the set of k true nearest neighbors, and let $A_k(q)$ be the set of the top k points returned by the algorithm. Then, recall@k is defined as:

$$\text{recall@}k = \frac{|P_k(q) \cap A_k(q)|}{k}$$

In our experiment as recall@5 we report an averaged value over all sequences of searches.

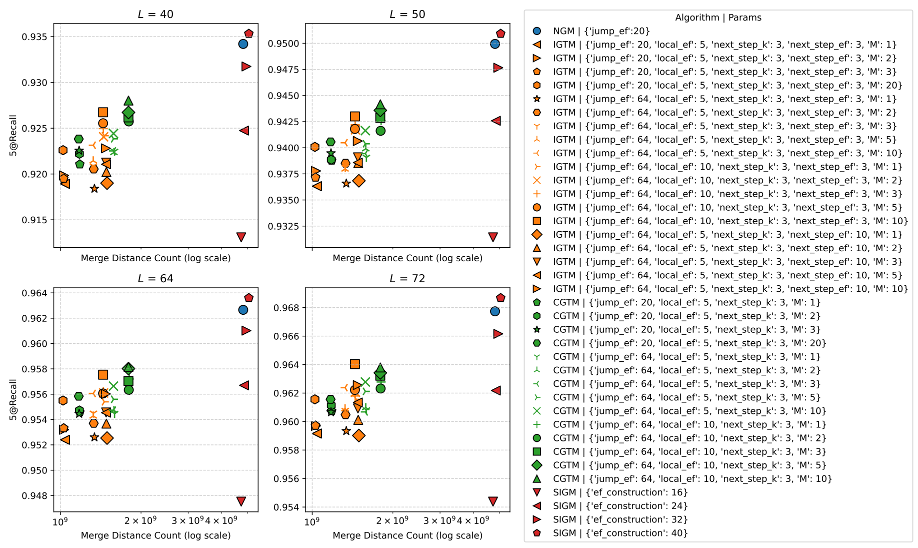

Figure 3: Recall of final merged graph vs merging efforts

5.2 Results

The searching quality of the merged graph is represented in Fig. 2. The graph plots trade-off between recall and the number of distance computations. As can be seen, SIGM with ef_construction ≥ 32 and NGM with parameter jump_ef $= 20$ produce merged graph with slightly better searing quality than other algorithms.

Another important fact which 2 shows that the number of distance computations performed by the search algorithm (Algorithm 2) for equal parameter L are almost the same. Therefore, the graph is better if it provides better search recall for a fixed value of the parameter L. Taking this into account, we can better interpret Fig. 3. It is easy to see the following:

- IGTM and CGTM in our experiment setup achieve recall better than SIGM with parameter ef_construciton $= 24$;
- NGM and SIGM perform the most computations, but achieves the highest recall, as it exhaustively reconstructs each neighborhood;
- CGTM achieving comparable recall with approximately 60% fewer computations than NGM and SIGM;
- IGTM provides the best run-time performance, reducing distance computations about 20% than CGTM, and about 70% than NGM and SIGM while achieving comparable recall.

6 Related Works on Graph Merging

While working on the paper, we have found that just recently researchers from **Elastic Search Lab** published a blog post [11] where they proposed a fast merge algorithm, which also utilizes information about closeness of object in all graphs. The authors presented a scheme where a subset of vertices J "join set" from the smaller graph is selected by greedy heuristic and inserted into the larger graph using standard HNSW insertion procedure. For the remaining vertices from the small graph, their already inserted neighbors and those neighbors' connections in the large graph are used to perform a limited beam search (named FAST-SEARCH-LAYER) to find connection candidates. This approach limits full insertions and accelerates the merging process.

Ideologically approach of [11] is similar to our work, because the authors utilize information about closeness of object in all graphs, and use light-weighted version of the search algorithm FAST-SEARCH-LAYER (in our case it is LOCAL SEARCH. The "join set" J from some point of view is similar to the set vertices for which IGTM and CGTM run the standard HNSW search. However, in [11] the "join set" J is determined separately at the first stage by a greedy heuristic that tries to find a good cover set, while IGTM and CGTM establish the vertices for which to run HNSW-Search during the work in the main cycle in a more homogeneous way.

The other work which is relatively close to the present topic is a paper by Zhao and co-authors [32]. They proposed two algorithms: Symmetric Merge (S-Merge) for combining two k-NN graphs and Joint Merge (J-Merge) for incrementally extending an existing graph with new points. These methods effectively combine graphs built on different subsamples while maintaining high search accuracy and accelerating computations. S-Merge discards half the neighbors in each list, while J-Merge leverages the hierarchical structure of the graph.

The performance comparison of the different merge algorithms is a subject of future research.

7 Conclusion and Future Work

In this work, we introduced three algorithms to merge hierarchical navigable small-world graphs: NGM (Algorithm 6), IGTM (Algorithm 7), and CGTM (Algorithm 8).

The NGM algorithm provides a straightforward but computationally intensive approach, performing standard HNSW searches to find sets of candidates to reconstruct the neighborhood of a vertex chosen in an arbitrary way. IGTM and CGTM improve efficiency by leveraging locality—processing vertices close to each other sequentially and using the less expensive LocalSearch algorithm (Algorithm 1).

Our experimental results demonstrate that IGTM and CGTM significantly reduce the number of distance computations compared to the naive approach while maintaining comparable search accuracy. Our evaluation on the dataset of 1 million 128-dimensional vectors (SIFT1M) shows that IGTM and CGTM are more than 3 times faster than the straightforward insertion strategy SIGM, with minimal impact on recall performance.

Interestingly, our experiments revealed that IGTM outperformed CGTM in terms of computational efficiency, contrary to our initial expectations. We had anticipated that CGTM would be more efficient since it can select the next vertex for neighborhood construction from both graphs, theoretically reducing the chances of getting stuck and thus requiring fewer costly standard search operations. However, it appears that the overhead of selecting the next processing vertex, for which a neighborhood is forming, in CGTM is too computationally expensive, and these costs are not offset by the reduction in searches when "gets stuck" with selecting the next close vertex to process. The study of this fact is a subject for further work.

In addition, an important direction for future work is adapting the proposed merge algorithms to handle deleted vertices. This would enable their use in compaction processes, where graphs are periodically restructured to remove obsolete entries and maintain search efficiency. By extending our merge algorithms to filter out deleted nodes during the "Processing Vertex Selection" phase.

Also, the researchers can focus on the idea that the neighborhood construction can be done for the set of vertex close to each other instead of forming neighborhood for one vertex per iteration. This can be more suitable for GPU or NPU settings.

Additional areas for exploration include adaptive parameter selection based on dataset characteristics, and extensions to other graph-based index structures beyond HNSW.

Finally, future work can be applied to develop a fast graph construction procedure using one of the merge algorithms. It seems very natural to start from the set of small graphs and merge them in a recursive manner.

References

[1] Yury Malkov et al. *hnswlib: Hierarchical Navigable Small World Graphs Library*. `https://github.com/nmslib/hnswlib`. Accessed: 2025-05-04. 2025.

[2] Alexandr Andoni and Piotr Indyk. "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions". In: *Communications of the ACM* 51.1 (2008), pp. 117–122.

[3] Alexandr Andoni and Ilya Razenshteyn. "Optimal Data-Dependent Hashing for Approximate Near Neighbors". In: *Proceedings of the 47th Annual ACM Symposium on Theory of Computing.* 2015, pp. 793–801.

[4] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. "ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms". In: *Information Systems* 87 (2020), p. 101374.

[5] Jon Louis Bentley. “K-d trees for semidynamic point sets”. In: *Proceedings of the sixth annual symposium on Computational geometry*. 1990, pp. 187–197.

[6] Alina Beygelzimer, Sham Kakade, and John Langford. “Cover trees for nearest neighbor”. In: *Proceedings of the 23rd international conference on Machine learning*. 2006, pp. 97–104.

[7] Leonid Boytsov and Bilegsaikhan Naidan. “Engineering Efficient and Effective Non-metric Space Library”. In: *Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings*. Ed. by Nieves R. Brisaboa, Oscar Pedreira, and Pavel Zezula. Vol. 8199. Lecture Notes in Computer Science. Springer, 2013, pp. 280–293. DOI: `10.1007/978-3-642-41062-8\_28`. URL: `https://doi.org/10.1007/978-3-642-41062-8%5C_28`.

[8] Paolo Ciaccia, Marco Patella, and Pavel Zezula. “M-tree: An E cient access method for similarity search in metric spaces”. In: *Proceedings of the 23rd VLDB conference, Athens, Greece*. Citeseer. 1997, pp. 426–435.

[9] Mayur Datar et al. “Locality-Sensitive Hashing Scheme Based on p-Stable Distributions”. In: *Proceedings of the 20th Annual Symposium on Computational Geometry*. 2004, pp. 253–262.

[10] Matthijs Douze et al. “The Faiss library”. In: (2024). arXiv: `2401.08281 [cs.LG]`.

[11] Elastic Team. *HNSW Graphs: Speed Up Merging*. 2023. URL: `https://www.elastic.co/search-labs/blog/hnsw-graphs-speed-up-merging#graph-merging` (visited on 04/05/2025).

[12] Tiezheng Ge et al. “Optimized Product Quantization for Approximate Nearest Neighbor Search”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2013, pp. 863–870.

[13] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. “Similarity Search in High Dimensions via Hashing”. In: *ACM SIGMOD Record* 28.2 (1999), pp. 517–528.

[14] Antonin Guttman. “R-trees: A dynamic index structure for spatial searching”. In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 1984, pp. 47–57.

[15] Piotr Indyk. “Local-Descriptor Matching for Image Identification”. In: *Proceedings of the Conference on High-Dimensional Hashing*. 1998.

[16] Suhas Jayaram Subramanya et al. “Diskann: Fast accurate billion-point nearest neighbor search on a single node”. In: *Advances in neural information processing Systems* 32 (2019).

[17] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. “Product Quantization for Nearest Neighbor Search”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (2011), pp. 117–128.

[18] Yu A Malkov and Dmitry A Yashunin. “Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs”. In: *IEEE transactions on pattern analysis and machine intelligence* 42.4 (2018), pp. 824–836.

[19] Yury Malkov et al. “Approximate nearest neighbor algorithm based on navigable small world graphs”. In: *Information Systems* 45 (2014), pp. 61–68.

[20] Yury Malkov et al. “Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces”. In: *Similarity Search and Applications: 5th International Conference, SISAP 2012, Toronto, ON, Canada, August 9-10, 2012. Proceedings 5*. Springer. 2012, pp. 132–147.

[21] Milvus Team. *HNSW Index*. 2024. URL: `https://milvus.io/docs/hnsw.md` (visited on 04/05/2025).

[22] Mohammad Norouzi and David J. Fleet. “Cartesian k-Means for Product Quantization”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2013, pp. 119–126.

[23] Oracle Corporation. *Understanding Hierarchical Navigable Small World Indexes*. Accessed: 2025-04-05. 2023.

[24] Alexander Ponomarenko et al. “Approximate nearest neighbor search small world approach”. In: *International Conference on Information and Communication Technologies & Applications*. Vol. 17. 2011.

[25] Liudmila Prokhorenkova and Aleksandr Shekhovtsov. “Graph-based nearest neighbor search: From practice to theory”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 7803–7813.

[26] Ash Vardanian. *USearch by Unum Cloud*. Version 2.17.7. Oct. 2023. DOI: `10.5281/zenodo.7949416`. URL: `https://github.com/unum-cloud/usearch`.

[27] Mengzhao Wang et al. “A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search”. In: *Proceedings of the VLDB Endowment* 14.11 (2021), pp. 1964–1978.

[28] Weaviate Team. *Vector Index*. Accessed: 2025-04-05. Weaviate. 2024.

[29] Shuo Yang et al. “Revisiting the index construction of proximity graph-based approximate nearest neighbor search”. In: *arXiv preprint arXiv:2410.01231* (2024).

[30] Peter N Yianilos. “Data structures and algorithms for nearest neighbor search in general metric spaces”. In: *Soda*. Vol. 93. 194. 1993, pp. 311–21.

[31] Pavel Zezula et al. *Similarity search: the metric space approach*. Vol. 32. Springer Science & Business Media, 2006.

[32] Wan-Lei Zhao et al. “On the merge of k-NN graph”. In: *IEEE Transactions on Big Data* 8.6 (2021), pp. 1496–1510.

[33] Zilliz Team. *Hierarchical Navigable Small World (HNSW)*. 2022. URL: `https://zilliz.com/learn/hierarchical-navigable-small-worlds-HNSW` (visited on 04/05/2025).