Three Algorithms for Merging Hierarchical Navigable Small World Graphs

Alexander Ponomarenko[1]

[1]HSE Univerity

1 Introduction

Navigable small-world graphs have emerged as a cornerstone for efficient approximate nearest neighbor (ANN) search in high-dimensional spaces. This problem is crucial in fields such as recommendation systems, image retrieval, and natural language processing. Algorithms like Hierarchical Navigable Small World (HNSW) [13] and its predecessor Navigable Small World (NSW) [17, 14, 15] provide robust and scalable frameworks for organizing data into graph structures that support fast similarity searches.

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

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

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

In high-dimensional spaces, this problem becomes computationally challenging due to the "curse of dimensionality." Exact solutions often degrade to linear scan as dimensionality increases, making them impractical for large-scale applications. This has led to the development of approximate k-nearest neighbor (ANN) search methods, which trade perfect accuracy for substantial improvements in computational efficiency.

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 [3], R-trees [10], VP-trees [18], M-trees [5], and Cover trees [4]. A comprehensive overview of these approaches for metric spaces is provided by Zezula et al. [19].

2. **Mapping-based methods**: These techniques transform the original data into representations that allow for efficient search. This category includes Locality-Sensitive Hashing (LSH) [11, 9, 6, 1, 2] and Product Quantization approaches [12, 8, 16], which encode vectors into compact codes that approximate 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 directed traversal through the data space.

1.2 Hierarchical Navigable Small World (HNSW)

HNSW [13] extends the navigable small-world concept by organizing points into a hierarchy of navigable graphs also named as layers. Each layer contains a progressively smaller subset of the data points. The bottommost layer contains all points, while higher layers form a sparse representation of the data space. Graphs of all levels glued together form a navigable small world graph like NSW [17]. An explicit separation small world graph to layers allows to reduce number of computation in search while traversing vertexes with high degrees also called hubs.

HNSW supports two primary operations: **search** (see section 2.2) and **insertion**. In the Fig. 1 is a schematic example of multilayered structure of HNSW.

1.3 Graph Merging as an Iterative Process

One of the key challenges in maintaining these graph-based indices is efficiently combining data from multiple sources. While HNSW supports dynamic insertions, merging entire graphs as coherent structures presents a different challenge. The quality of the merged graph directly impacts search performance, and naive approaches may lead to suboptimal neighborhood connections or excessive computational costs during the merge operation.

It is worth noting that graph merging can be used as an analog to compaction operations in database systems. 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. Moreover, a true deletion might trigger cascading neighborhood reconstructions that could potentially affect the entire graph, which is

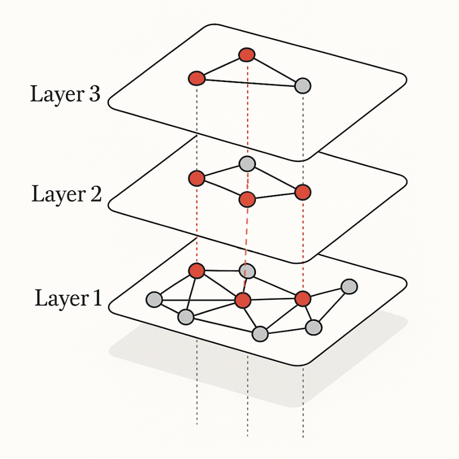

Figure 1: Schematic example HNSW structure. Each layer is a navigable graph that suit for "greedy-like" traversal algorithms. Graphs of all levels glued together form a navigable small world graph like similar to NSW. An explicit separation small world graph to layers allows to reduce number of computation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

unacceptable from a performance perspective. Graph merging provides a way to periodically compact the structure by creating a fresh graph that excludes deleted vertices.

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 navigable 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, promoting locality within original graphs. In contrast, CGTM implements the idea that the next vertex for neighborhood construction can be selected from any input graph, enabling more thorough exploration of the combined space. While our methods primarily target HNSW, they are generalizable to other graph-based data structures such as NSW [17] and NSG [7]. These algorithms preserve graph navigability and maintain the structural integrity necessary for fast ANN search while offering different trade-offs between merge accuracy and computational efficiency.

The rest of this paper is organized as follows: Section 2 describes the search algorithms for navigable graphs that we use in merge procedures. Section 3 presents 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 experimental results comparing the proposed methods. Finally, Section 6 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

The LocalSearch algorithm (Algorithm 1) implements a greedy 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.

For each iteration, LocalSearch selects the nearest 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 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

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 $V^* \subset V$

```
while True do
    $u \leftarrow$ nearest unvisited point to $q$ in $C$
    $U \leftarrow \{v \mid (u, v) \in E\}$
    for $v \in U$ do
        if $v$ is unvisited then
            $C \leftarrow C \cup \{v\}$
    if $|C| > L$ then
        $C \leftarrow$ top $L$ nearest points to $q$ in $C$
    if no updates to $C$ then
        break
return top-$k$ nearest points to $q$ in $C$
```

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 broader exploration, potentially escaping local minima and finding better global solutions, but at the expense of increased computation time. In the HNSW paper, parameter L 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 $V^* \subset V$

```
$v^* \leftarrow v_0$
for $i = l_{\max}$ down to $\ell$ do
    $v^* \leftarrow$ LocalSearch$(G = G_i, q = q, C = \{v^*\}, k = 1, L = L)$
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. 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 the entry point for the subsequent layer. At the target layer ℓ, a final LocalSearch is performed with the specified k value 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.

The parameters k, L, and the choice of entry point v_0 collectively influence the search quality and efficiency. Typically, v_0 is selected as the entry point at the highest layer of the HNSW structure, though any vertex in the graph can theoretically serve as the starting point.

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.

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
$C' \leftarrow k$-nearest neighbors of $v^*$ in $C$
return $C'$
```

Various strategies exist for selecting a node's neighborhood in navigable small-world graphs. 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 graph density.

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.

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

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

```
Sort $C$ in ascending order by distance to $v^*$
$C' \leftarrow \emptyset$
for $v \in C$ do
    $f \leftarrow$ true
    for $w \in C'$ do
        if $\rho(v^*, v) \geq \rho(v, w)$ then
            $f \leftarrow$ false
            break
    if $f$ then
        $C' \leftarrow C' \cup \{v\}$
    if $|C'| \geq m$ then
        break
return $C'$
```

4 Merge Algorithms

We now present three algorithms for merging HNSW graphs: SIGM, 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 provide a short descriptopn of simple insertion graph merge (SIGM). It is a default merging strategy that uses an insertion operation to merge graphs. It takes the largest graph and sequentially inserts data from other graph. Actually, it is not 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 efficient 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 $\mathcal{H}_c = (G_i^c)_{i=0}^{l_{\max}^c}$
$l_{\max}^a \leftarrow \mathcal{H}_a$.getMaxLayerNumber()
$l_{\max}^b \leftarrow \mathcal{H}_b$.getMaxLayerNumber()
$l_{\max}^c \leftarrow \max(l_{\max}^a, l_{\max}^b)$
for $i = 0$ to $l_{\max}^c$ do                                          ▷ Merge each layer
    $G_i^c \leftarrow$ LayerMergeAlgo($\mathcal{H}_a, \mathcal{H}_b, \ell = i$, params)   ▷ Calls e.g., Merge-Naive
$\mathcal{H}_c \leftarrow (G_i^c)_{i=0}^{l_{\max}^c}$
return $\mathcal{H}_c$
```

Note that the merging of different layers (iterations of the loop in Algorithm 5) is independent and can potentially be parallelized.

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_ℓ^a (line 5), the algorithm:

1. Searches for potential neighbors in graph G_ℓ^b using HNSW-Search (line 6)

2. Combines these candidates with v^*'s original neighbors from G_ℓ^a (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 process is then repeated for all vertices of graph G_ℓ^b (lines 9-12), searching for their potential neighbors in G_ℓ^a.

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($\mathcal{H}_a, \mathcal{H}_b, \ell, \text{NeighborhoodConstruction}, m, \text{search\_ef}, v_{entry}$)

```
Input: HNSW graphs $\mathcal{H}_a$, $\mathcal{H}_b$; target layer $\ell$; 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: $G^a \leftarrow \mathcal{H}_a.\text{GetLayer}(\ell)$; $G^b \leftarrow \mathcal{H}_a.\text{GetLayer}(\ell)$
2: $V^a \leftarrow \text{vertices}(G^a)$; $V^b \leftarrow \text{vertices}(G^b)$
3: $E^a \leftarrow \text{edges}(G^a)$; $E^b \leftarrow \text{edges}(G^b)$
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($\mathcal{H} = \mathcal{H}_b, q = v^*, v_{entry} = v^b_{entry}, k = m, L = \text{search\_ef}, \ell = \ell$)
7:     $\mathcal{C} \leftarrow \{v \mid (v^*, v) \in E^a\} \cup \mathcal{C}^b$
8:     $E^c \leftarrow E^c \cup \{(v^*, v) | v \in \text{NEIGHBORHOOD\_CONSTRUCTION}(\mathcal{C}, v^*, m)\}$
9: for $v^* \in V^b$ do
10:    $\mathcal{C}^a \leftarrow$ HNSW-SEARCH($\mathcal{H} = \mathcal{H}_a, q = v^*, v_{entry} = v^a_{entry}, k = m, L = \text{search\_ef}, \ell_{target} = \ell$)
11:    $\mathcal{C} \leftarrow \{v \mid (v^*, v) \in E^b\} \cup \mathcal{C}^a$
12:    $E^c \leftarrow E^c \cup \{(v^*, v) | v \in \text{NEIGHBORHOOD\_CONSTRUCTION}(\mathcal{C}, v^*, m)\}$
13: return $G^c = (V^c, E^c)$
```

4.4 Intra Graph Traversal Merge

The most effort of NGM algorithm lies in the set of neighborhood candidates from the opposite graph utilising HNSW-SEARCH procedure, that traverses layers graph from the top-level down to layer number ℓ every time. The number of computations can be reduced if we select the next vertex v^* close to the previous one (line 15), instead of choosing it randomly. Thus, for new v^* the neighborhood candidates will be also close to the previous candidates set. To search this new neighborhood candidates we can use LOCALSEARCH procedure, that traversing the same graph staring from the previous neighborhood candidates set $\mathcal{P}^b$ (line 11,14). In the line 14 in set $\mathcal{P}^b$ we keep only M-closest to v^* candidates.

In the line 15 almost at each iteration we try select new v^* vertex close to the previous v^* that was not already processed. To ensure that new v^* vertex are not very far from the previous v^* we bound the size of results of LOCALSEARCH procedure controlling by "next_step_k" parameter. Once LOCALSEARCH can't find enough close not processed vertex, in line 7 we choice new v^* from the set $\mathcal{V}_{not\_done}$ randomly.

After we have processed all vertices from the the graph G^a. We do the same for vertices of graph G^b (lines 21-22).

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

```
$G^a \leftarrow \mathcal{H}_a.\text{GetLayer}(\ell)$; $G^b \leftarrow \mathcal{H}_a.\text{GetLayer}(\ell)$
$V^a \leftarrow \text{vertices}(G^a)$; $V^b \leftarrow \text{vertices}(G^b)$
$E^a \leftarrow \text{edges}(G^a)$; $E^b \leftarrow \text{edges}(G^b)$
$V^c \leftarrow V^a \cup V^b$; $E^c \leftarrow \emptyset$
$\mathcal{V}_{not\_done} \leftarrow V^a$
while $\mathcal{V}_{not\_done} \neq \emptyset$ do
    $v^* \leftarrow$ random choice from $\mathcal{V}_{not\_done}$
    $\mathcal{P}^b \leftarrow \text{HNSW-Search}(\mathcal{H} = \mathcal{H}^b, q = v^*, v_0, k = M, L = \text{jump\_ef}, \ell)$
    while True do
        $\mathcal{V}_{not\_done} \leftarrow \mathcal{V}_{not\_done} \setminus \{v^*\}$
        $\mathcal{C}^b \leftarrow \text{LocalSearch}(G = G^b, q = v^*, C = \mathcal{P}^b, k = m, L = \text{local\_ef})$
        $\mathcal{C} \leftarrow \{v : (v^*, v) \in E^a\} \cup \mathcal{C}^b$
        $E^c \leftarrow E^c \cup \{(v^*, v) : v \in \text{NeighborhoodConstruction}(\mathcal{C}, v^*, m)\}$
        $\mathcal{P}^b \leftarrow \{\mathcal{C}_1^b, \mathcal{C}_2^b, \ldots, \mathcal{C}_M^b\}$
        $\mathcal{C}^a \leftarrow \text{LocalSearch}(G = G^a, q = v^*, C = \{v^*\}, k = \text{next\_step\_k}, L = \text{next\_step\_ef})$
        $\mathcal{C}^a \leftarrow \mathcal{C}^a \cap \mathcal{V}_{not\_done}$
        if $\mathcal{C}^a = \emptyset$ then
            break
        $v^* \leftarrow \mathcal{C}_1^a$
$\mathcal{V}_{not\_done} \leftarrow V^b$
while $\mathcal{V}_{not\_done} \neq \emptyset$ do
    Repeat the same process for $V^b$ with the roles of $\mathcal{H}_a$ and $\mathcal{H}_b$ swapped.
return $G^c = (V^c, E^c)$
```

4.5 Cross Graph Traversal Merge

Cross Graph Traversal Merge (CGTM) algorithm is similar to IGTM utilise LocalSearch procedure to reduce computation effort. The difference is that IGTM algorithm the next v^* vertex choice from the same graph, while CGTM looks for new vertex v^* in the both graphs G^a, and G^b. Thus in line 24 v^* is chosen from the set $\mathcal{C}_{not\_done}$, which of not processed vertex from the both graph (line 21). The intuition laying behind CGTM algorithm is that allowing to choice new v^* from the both graphs, we reduce the number of times what v^* is chosen randomly, thereby minimise the number of using more expensive search procedure HNSW-Search.

Algorithm 8 CGTM($\mathcal{H}_a, \mathcal{H}_b, \ell$, jump_ef, local_ef, 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($\mathcal{H} = \mathcal{H}^a, q = v^*, v_0, k, L =$ jump_ef, $\ell$)
 9:     $\mathcal{P}^b \leftarrow$ HNSW-Search($\mathcal{H} = \mathcal{H}^b, q = v^*, v_0, k, L =$ 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($G = G^a, q = v^*, C = \mathcal{P}^a, k = m, L =$ local_ef)
13:         $\mathcal{C}^b \leftarrow$ LocalSearch($G = G^b, q = v^*, C = \mathcal{P}^b, k = m, L =$ 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:         $E^c \leftarrow E^c \cup \{(v^*, v) : v \in$ 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 $G^c = (V^c, E^c)$
```

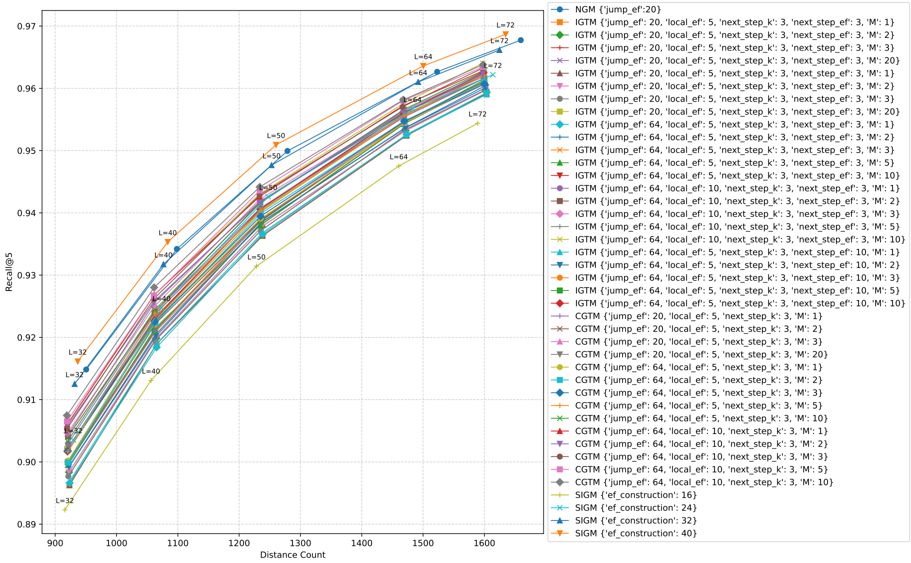

Figure 2: Recall vs distance count on search stage for merged graphs

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. On those subsets we built two HNSW indices $\mathcal{H}_a$ and $\mathcal{H}_b$ using the following parameters: $M = 16$, $M0 = 32$, ef_construction = 32. These indices are then merged using simple insertion stategy textscSIGM and the three proposed algorithms: textscNGM, IGTM, and CGTM. We keep the same values of parameters $m = 16$, and $m0 = 32$ as the target neighborhood size in the merged graph.

In order to be independent of the implementation details and low level optimisation, we have done comparison of the algorithms based on the number of distance computation required for the merge process. The other important characteristics of merging process is how accurate the merge is done, i.e. 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 top k points returned by the algorithm.

Then, the 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 searches.

5.2 Results

The searching quality of the merged graph can be seen from the Fig. 2. The graph plots trade-off between recall and the number of distance computations. As can be seen, SIGM with ef_construction $\geq$ 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 searching 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 IGTM 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 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 both IGTM and CGTM significantly reduce the number of distance computations compared to the naive approach while maintaining comparable search accuracy. Our evaluation on the

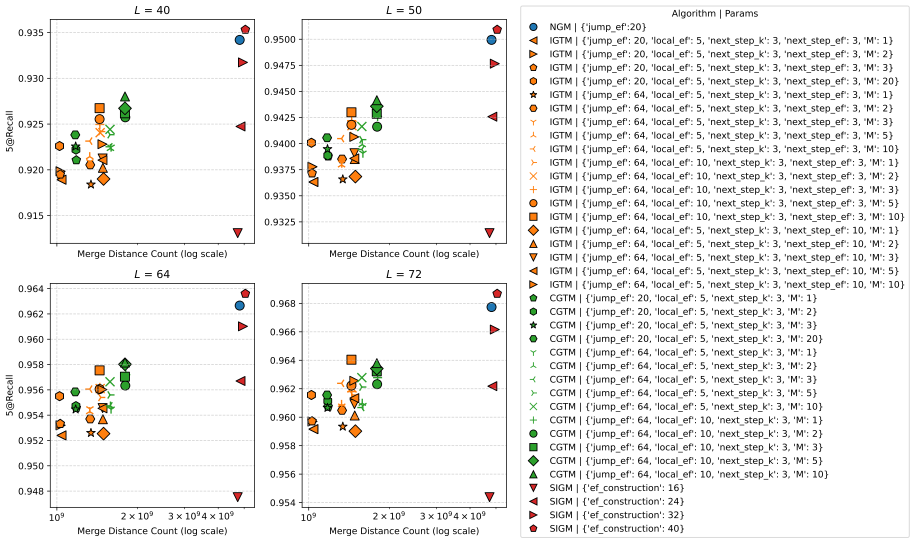

Figure 3: Recall of final merged graph vs merging efforts

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 "getting 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.

Finally, 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.

References

[1] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117–122, 2008.

[2] 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*, pages 793–801, 2015.

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

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

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

[6] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th Annual Symposium on Computational Geometry*, pages 253–262, 2004.

[7] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

[8] Tiezheng Ge, Kaiming He, Qifeng Chen, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 863–870, 2013.

[9] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. *ACM SIGMOD Record*, 28(2):517–528, 1999.

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

[11] Piotr Indyk. Local-descriptor matching for image identification. In *Proceedings of the Conference on High-Dimensional Hashing*, 1998.

[12] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[13] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[14] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 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*, pages 132–147. Springer, 2012.

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

[16] 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)*, pages 119–126, 2013.

[17] Alexander Ponomarenko, Yury Malkov, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor search small world approach. In *International Conference on Information and Communication Technologies & Applications*, volume 17, 2011.

[18] Peter N Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Soda*, volume 93, pages 311–21, 1993.

[19] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity search: the metric space approach*, volume 32. Springer Science & Business Media, 2006.