On the Merge of k-NN Graph

Wan-Lei Zhao*, Hui Wang, Peng-Cheng Lin and Chong-Wah Ngo, *Senior Member, IEEE*

Abstract—k-nearest neighbor graph is a fundamental data structure in many disciplines such as information retrieval, data-mining, pattern recognition, and machine learning, etc. In the literature, considerable research has been focusing on how to efficiently build an approximate k-nearest neighbor graph (k-NN graph) for a fixed dataset. Unfortunately, a closely related issue of how to merge two existing k-NN graphs has been overlooked. In this paper, we address the issue of k-NN graph merging in two different scenarios. In the first scenario, a symmetric merge algorithm is proposed to combine two approximate k-NN graphs. The algorithm facilitates large-scale processing by the efficient merging of k-NN graphs that are produced in parallel. In the second scenario, a joint merge algorithm is proposed to expand an existing k-NN graph with a raw dataset. The algorithm enables the incremental construction of a hierarchical approximate k-NN graph. Superior performance is attained when leveraging the hierarchy for NN search of various data types, dimensionality, and distance measures.

Index Terms—k-nearest neighbor graph, nearest neighbor search, high-dimensional, k-NN graph merge

✦

1 INTRODUCTION

Given a dataset $S = \{x | x \in R^d\}$ with n samples, k-NN graph refers to the data structure, $G[i]$, that keeps the top-k nearest neighbors of every sample x_i in the dataset. It is a key data structure in the manifold learning [1], [2], data mining, machine learning [3], and information retrieval. It is also supportive for balanced graph construction [4]. Basically, given a metric $m(\cdot, \cdot)$, the construction of k-NN graph is to find the top-k nearest neighbors for each data sample. When being built in a brute-force way, the time complexity is $O(d \cdot n^2)$, where d is the dimension and n is the size of a dataset. As the value of n can be very large in big data applications, the construction of an exact k-NN graph is computationally expensive. The problem is even challenging if the value of d is also large, *i.e.*, the samples are in high dimensional space. Due to its high time complexity, the existing works [5], [6], [7], [8] instead aim for an efficient but approximate solution.

Despite constant progress has been made in the last few decades, to the best of our knowledge, there is no research dedicated for merging k-NN graphs. Given two k-NN graphs G_1 and G_2 from the datasets $S_1 = \{x | x \in R^d\}$ and $S_2 = \{y | y \in R^d\}$ respectively, the task of merging is to build G for $S = S_1 \cup S_2$ based on G_1 and G_2 instead of reconstructing G from scratch[1]. During merging, G_1 is updated with the nearest neighbor samples discovered on S_2. A similar process is carried out to update G_2 with S_1.

The problem is important for enlightening large-scale construction of approximate k-NN graph in an incremental manner. For a fixed dataset, merging ensures an efficient divide-and-conquer way of graph construction. In some applications, k-NN graphs can only be built for individual subsets on the initial stages, given the data are distributed on different nodes or not all of them are ready at that moment. In the case of parallel computing, one would prefer slicing the data into blocks and computing the approximate k-NN graph for each block on different machines. The subgraphs are later reduced into one by repeatedly merging two subgraphs each time. For the online data streamed in batch, merging is a fundamental step to expand an already-built k-NN graph with new raw samples. The step naturally extends an approximate k-NN graph construction algorithm to online version. The typical scenario takes place in a multimedia website such as Flickr, where the k-NN graph is built to support content-based photo search. In this scenario, the constructed k-NN graph should be merged with new nodes when new photos are uploaded from time to time. Considering the big number of photos and the high upload frequency in the website, the re-construction of the graph from scratch from time to time is nearly impossible. This is where the k-NN graph merge algorithm could fit in.

A brute force way of merging graph G_1 with graph G_2 can be undertaken by a thorough cross-comparison between samples in S_1 and samples in S_2. The produced edge, say $< x_i, y_j, m(x_i, y_j) >$, is inserted into $G_1[i]$ and $G_2[j]$ if it is ranked at top-k of the corresponding NN list. Given the cardinalities of S_1 and S_2 are n_1 and n_2 respectively, the time complexity of such cross-comparison is $O(n_1 \cdot n_2 \cdot d)$, which is even higher than constructing the graph from scratch by algorithms such as [5], [6]. Hence, the problem of merging is not trivial.

In this paper, two efficient and generic algorithms for k-NN graph merge are proposed. They are designed to address the merging issues under two different scenarios. In the first scenario, we address the issue of merging two already-built k-NN graphs. Merging more than two graphs can be trivially implemented with a recursive call of two-graph merging. In the second scenario, we address the issue of merging a raw sample set into an already built k-NN

- *Wan-Lei Zhao, Peng-Cheng Lin and Hui Wang are with the Department of Computer Science, Xiamen University, Xiamen, China. Wan-Lei Zhao is the corresponding author. E-mail: wlzhao@xmu.edu.cn*

- *Chong-Wah Ngo is with School of Computing and Information Systems, Singapore Management University.*

1. Without the loss of generality, we assume $S_1 \cap S_2 = \emptyset$.

graph. The merging algorithm novelly leads to a hierarchical approximate k-NN graph construction algorithm. Capitalizing on the hierarchical structure, an efficient nearest neighbor (NN) search algorithm is presented.

The remainder of this paper is organized as follows. In Section 2, a brief review of the research works on approximate k-NN graph construction and NN search is presented. Section 3 presents two algorithms for k-NN graph merge under two different scenarios. They are called symmetric merge (S-Merge) and joint merge (J-Merge). Deriving from J-Merge, a hierarchical approximate k-NN graph construction algorithm is presented at the end of this section. In Section 4, based on the constructed hierarchical approximate k-NN graphs, an efficient NN search approach is presented. Section 5 presents empirical findings to justify the effectiveness of the proposed algorithms for k-NN graph merging and NN search. Section 6 concludes the paper.

2 RELATED WORKS

2.1 Approximate k-NN Graph Construction

In the approximate k-NN graph construction, there are in general two types of approaches. The space partitioning approaches such as [7], [8], [9] follow a two-stage pipeline. At the first stage, samples are divided into a number of small subsets by random partitions. The random partitions over the space can be generated using techniques such as recursive Lanczos bisection [7], hierarchical random projections [8], or a series of locality-sensitive hash functions [9]. The samples in each subset are expected to be close to each other. Exhaustive pairwise comparisons are carried out within each subset. The closeness relations (*viz.*, edges in the k-NN graph) between any two samples in one subset are established. Based on the closeness relations, the second stage selects samples to update the k-NN graph. Specifically, these samples substitute the existing samples with longer edges in the k-NN graph. The two stages iterate and perform updates incrementally. Due to the difficulty in designing partitioning schemes generic for various metric spaces, this type of approach is generally effective in l_p-space only.

NN-Descent [5], the second type of k-NN graph construction, differs by its ability to operate under various distance measures. The algorithm starts the construction from a random k-NN graph. Based on the principle of "neighbor's neighbor is likely to be the neighbor", the cross-comparison is invoked for the samples staying in a neighborhood region. The closer pairs of samples are then selected to update the neighborhood of a target sample. The iteration continues until no change of neighborhood configuration. As the algorithm essentially performs hill-climbing optimization batchfully [5], [10] independent of metric space, it is both generic and efficient. Recently, the hybrid scheme based on both the space partitioning approach and NN-Descent has been proposed in the literature [6]. Although efficient, this hybrid scheme is infeasible for the metrics beyond l_p-norms.

Other efficient approaches include KIFF [11] which is specifically designed to build k-NN graph for the high-dimensional sparse dataset. The k-NN graph is built by sequentially considering the sample pairs with high co-occurrence frequency. This approach turns out to be much more efficient than NN-Descent [5] on sparse datasets.

All the aforementioned approaches assume k-NN graph construction under static dataset. To cope with streaming data, such as photos and videos uploaded on a daily basis to the social media platform, an incremental update is demanded. Nevertheless, there is no such incremental updating strategy being considered in these algorithms. Although NN-Descent can be applied to augment an already-built k-NN graph with new samples, the data aggregation cost is high if updates are performed frequently.

Online graph construction is proposed in [12], [13] primarily for the nearest neighbor search. These approaches, nevertheless, are not designed to build a k-NN graph. In order to increase search efficiency, the k-NN graph is sparsified during the construction. Consequently, the right samples supposedly to be included in the k-NN list are deliberately omitted when being occluded by other neighboring samples. Instead, the links to remote neighbors are maintained. During the search, these links serve as short cuts to allow quick access to the close neighborhood of a query. However, due to the amendment, the graph is no longer a k-NN graph by definition and does not support tasks beyond NN search.

This paper addresses the problem of merging k-NN graphs in two typical scenarios. In our solution, the merge operation enables the dynamic construction of k-NN graph, through the incremental update of the graph with the raw dataset. Moreover, like NN-Descent [5], our algorithm is generic to various distance measures.

2.2 NN Search

An issue that is closely related to approximate k-NN graph construction is the nearest neighbor search. The primary goal of NN search is to find out the nearest neighbors from a given dataset for a query sample. In the problem, both the query and the candidate samples are assumed to be from the same space *i.e.*, R^d.

This issue has been traditionally addressed by a variety of tree partitioning approaches, such as K-D tree [14], R-tree [15], X-Tree [16], NV-tree [17] and Trinary-Projection tree [18]. The idea is to partition a space into hierarchical sub-spaces such that only a few branches of sub-spaces are traversed during NN search. However, unlike B-tree in 1D case, the true nearest neighbor may reside in the branches that are outside the candidate sets. Therefore, extensive probing over a large number of branches in the tree becomes inevitable. Recent indexing structures FLANN [19] and Annoy [20] partition the space with hierarchical k-means and multiple K-D trees respectively. Although both of them are efficient, sub-optimal results are achieved.

Apart from tree partitioning approaches, the quantization based approaches [21], [22], [23], [24] and locality-sensitive hashing (LSH) [25], [26], [27], [28], [29] have been extensively explored in the last decade. Compared to tree partitioning, these approaches are efficient in terms of both memory and speed. The speedup is significant especially when the size of encoding bits is short. Nevertheless, these advantages are traded off by relatively lower search accuracy. In addition, these approaches are mostly only suitable for l_p-norms. The design of generic hash functions is non-trivial.

Recently, the graph-based approaches such as hill-climbing [10] and nearest neighbor descent (NN-Descent) [5], demonstrate superior performance over other categories of approaches in many large-scale NN search tasks [13], [30], [31], [32]. All the approaches in this category are built upon an approximate k-NN graph or diversified approximate k-NN graph. The search procedure starts from a group of random seeds and traverses iteratively over the graph by the best-first search. Guided by the neighbors of visited vertices, the search procedure descents closer to the true nearest neighbor in each round until no better candidates could be found. The approaches in [6], [10], [12], [13], [32], [33], [34] in general follow a similar search procedure. The major difference between them lies in the structure of graphs upon which the NN search is undertaken. For most of the graph-based approaches, the space complexity is roughly linear to the scale of a dataset. And the extra merit is that they are suitable for various distance measures.

In this paper, a hierarchical approximate k-NN graph construction algorithm, as a by-product of graph merging operation, is derived for NN search. Similar to HNSW [13], the hierarchical graph achieves speedup by skipping samples lying in the far neighborhood of a query. The speedup is particularly significant in low dimensional space. Compared to HNSW, the algorithm offers relatively better search performance due to the high quality of the approximate k-NN graph. Furthermore, as the intermediate result of graph construction, the hierarchical structure is not required to be in memory during construction. The burden in maintaining and updating the hierarchy is therefore relieved. Moreover different from HNSW, the produced graph structure also facilitates other tasks such as browsing over close neighbors of each sample (e.g., photos) since it maintains an approximate k-NN graph.

3 TWO k-NN GRAPH MERGE STRATEGIES

In this section, two schemes that are used to merge the k-NN graphs in two different scenarios are developed. In the first scenario, it is assumed that two sub-k-NN graphs are already built by any existing algorithms such as [5], [6], [32] or in a brute-force way. The algorithm merges two subgraphs into one. In the second scenario, the merge algorithm deals with the problem of joining a raw set into an already built k-NN graph. Without loss of generality, we assume that there is no intersection between the two subsets to be merged in both scenarios. Finally, a hierarchical approximate k-NN graph construction algorithm is proposed as a derivation from the second merge scheme.

3.1 Symmetric Merge

Given datasets $S_1 = \{s_i | s_i \in R^d\}$ and $S_2 = \{s_j | s_j \in R^d\}$ ($S_1 \cap S_2 = \emptyset$), k-NN graphs G_1 and G_2 have been built for S_1 and S_2 respectively. Now we consider the problem of constructing k-NN graph G for dataset $S = S_1 \cup S_2$ based on G_1 and G_2. In our solution, we propose to merge the graphs in three major steps, which are detailed in Alg. 1.

In the first step, the rear $\frac{k}{2}$ samples in each k-NN list from G_1 and G_2 are truncated out (*Lines 1-2*). These truncated lists are kept for later use. After the truncation, there are $\frac{k}{2}$

Algorithm 1: Symmetric Merge (S-Merge)

Data: S: dataset; G_1: k-NN Graph of S_1 of size m; G_2: k-NN Graph of S_2 of size n

Result: G: k-NN Graph G

1 **Divide** G_1 into G_1^u and G_1^v;
2 **Divide** G_2 into G_2^u and G_2^v;
3 **for** *i=1; i ≤ m; i++* **do**
4   **Append** $G_1^u[i]$ with $\frac{k}{2}$ random samples from S_2;
5 **end**
6 **for** *i=1; i ≤ n; i++* **do**
7   **Append** $G_2^u[i]$ with $\frac{k}{2}$ random samples from S_1;
8 **end**
9 $G \leftarrow G_1^u \cup G_2^u$;
10 **repeat**
11     $\overline{G} \leftarrow Reverse(G)$;
12     $\mathcal{G}[i] \leftarrow \overline{G}[i] \cup G[i], \forall i \in G$;
13     $c \leftarrow 0$;
14   **for** $u \in \mathcal{G}$ **do**
15   **for** $s_i, s_j \in \mathcal{G}[u], s_i \neq s_j$ **do**
16   **if** $s_i \in S_1$ & $s_j \in S_2$ *or* $s_i \in S_2$ & $s_j \in S_1$ **then**
17              $l \leftarrow m(s_i, s_j)$;
18   UpdateNN($G[s_i], s_j, l$);
19   UpdateNN($G[s_j], s_i, l$);
20              $c \leftarrow c + 1$;
21   **end**
22   **end**
23 **until** $c == 0$;
24 $G \leftarrow$ **Merge and Sort** G with G_1^v and G_2^v ;
25 **return** k-NN Graph G

---

samples in each NN list of G_1 and G_2. Each NN list of the graphs is then appended with $\frac{k}{2}$ samples randomly drawn from a different dataset. For instance, G_1^u is appended with the samples from S_2 (*Lines 3-5*) and similarly for G_2^u (*Lines 6-8*). Combining the augmented G_1^u and G_2^u leads to a half-baked k-NN graph G for the set $S = S_1 \cup S_2$ (*Line 9*). In the second step, the NN-Descent iteration [5] is performed on each NN list of graph G. The iteration continues until it converges (*Lines 10-23*). In *Line 11*, the function $Reverse(G)$ returns a reverse graph for the graph G [5], which keeps the reverse NN list for each sample. In *Lines 18-19*, function $UpdateNN(G[s_i], s_j, l)$ inserts a new edge $< s_i, s_j, l >$ into the NN list of s_i. Function $UpdateNN$ takes the k-NN list of s_i, sample ID of s_j and the distance l between s_i and s_j as the input parameters.

Different from [5], the cross-comparison on each k-NN list only takes place between samples from two different sets, namely between $s_i \in S_1$ and $s_j \in S_2$. Finally, the truncated rear lists from G_1 and G_2 are combined with G by a simple merge sort on each k-NN list. The top-k samples in each NN list are kept (*Line 24*) after the merge sort. Then the approximate k-NN graph for the whole set S is constructed. The merge algorithm is illustrated in Fig. 1.

Since the merge is undertaken on two constructed sub-k-NN graphs, the merge algorithm is called symmetric merge (S-Merge) from now on. In the first step of S-Merge, 50% of samples in each k-NN list of graphs G_1 and G_2 are cut out for the final merge. Whether to reserve more or fewer
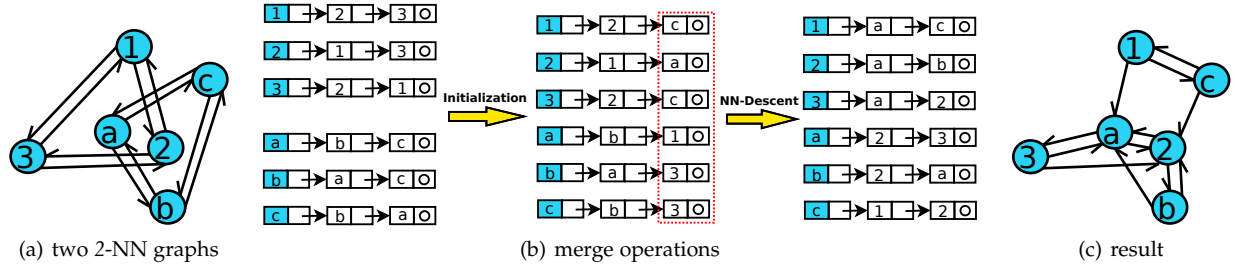
Fig. 1: The illustration of symmetric merge (S-Merge). Figure (a) shows two disconnected subgraphs. For distinction, the vertices of two subgraphs are denoted as numbers $S_1 = \{1, 2, 3\}$ and alphabets $S_2 = \{a, b, c\}$ respectively. Each subgraph is a 2-NN graph. Figure (b) illustrates the three steps in S-Merge. Figure (c) is the resulting graph after S-Merge.

than *50%* of neighbors in a *k*-NN list is an empirical setting. As will be shown in the ablation analysis (Section 5.2.1), a merged graph is maintained in high quality when the proportion of samples being kept is in the range of $[\frac{k}{5}, \frac{k}{2}]$.

S-Merge is particularly helpful for building an approximate *k*-NN graph in parallel. Imagine that multiple subgraphs are built with different threads or on different nodes by any existing algorithms [5], [32]. S-Merge is repeatedly called whenever any two subgraphs are ready to be merged. The process continues until a complete *k*-NN graph for the entire dataset is constructed.

3.2 Joint Merge

Based on a similar idea as S-Merge, the joint merge algorithm deals with the problem of merging a raw set into a built *k*-NN graph. Given two datasets $S_1 = \{s_i | s_i \in R^d\}$ and $S_2 = \{s_j | s_j \in R^d\}$, only the *k*-NN graph G_1 for S_1 is available. The problem is to build a graph G for dataset $S = S_1 \cup S_2$. Intuitively, exhaustive comparisons could be performed between the samples from S_1 and S_2 and within S_2. In this way, the *k*-NN lists in graph G_1 are updated while the *k*-NN lists for S_2 are established. Unfortunately, the time complexity of this way is high. Another possible solution is to construct a *k*-NN graph for S_2 by NN-Descent and perform S-Merge between the two subgraphs. It is feasible, whereas, not the most cost-effective solution. Following a similar idea as S-Merge, the joint merge (J-Merge) is proposed to fulfill the merge. J-Merge algorithm is summarized in Alg. 2.

The *k*-NN graph G_1 is initially cut into G_1^u and G_1^v. G_1^u keeps the top $\frac{k}{2}$-NN lists of G_1, while G_1^v maintains the remaining lists in G_1 for merge sort in the final stage. The NN lists of G_1^u are appended with $\frac{k}{2}$ samples randomly drawn from S_2. For dataset S_2, a random *k*-NN list is initialized for each sample. The samples are selected from S (*i.e.*, the union of S_1 and S_2). This raw graph is denoted as G_2. Combining with G_1^u, we have a half-baked *k*-NN graph G for the set S. The iteration continues until no new update happens. Finally, the truncated G_1^v is combined with G by merge sort on each *k*-NN list. The approximate *k*-NN graph for the whole set G is produced by keeping the top-*k* samples in each NN list. Similar to S-Merge, the proportion of samples to be chopped from a *k*-NN list of G_1 is an empirical setting. As will be verified in the experiment, the proportion in the range of $[\frac{k}{3}, \frac{k}{2}]$ suffices for a graph with

---

Algorithm 2: Joint Merge (J-Merge)

Data: S: dataset; G_1: *k*-NN Graph of S_1 of size m; S_2: raw dataset of size n

Result: G: *k*-NN Graph G

1 **Divide** G_1 into G_1^u and G_1^v;
2 **for** *i=1; $i \leq m$; i++* **do**
3   **Append** $G_1^u[i]$ with $\frac{k}{2}$ random samples from S_2;
4 **end**
5 **for** *i=1; $i \leq n$; i++* **do**
6   **Initialize** $G_2[i]$ with k random samples from $S_1 \cup S_2$;
7 **end**
8 $G \leftarrow G_1^u \cup G_2$;
9 **repeat**
10    $\overline{G} \leftarrow Reverse(G)$;
11    $\mathcal{G}[i] \leftarrow \overline{G}[i] \cup G[i], \forall i \in G$;
12    $c \leftarrow 0$;
13   **for** $u \in \mathcal{G}$ **do**
14   **for** $s_i, s_j \in \mathcal{G}[u], s_i \neq s_j$ **do**
15   **if** $s_i \in S_1$ *&* $s_j \in S_2$ *or* $s_i \in S_2$ *&* $s_j \in S_1$ *or* $s_i, s_j \in S_2$ **then**
16          $l \leftarrow m(s_i, s_j)$;
17   UpdateNN($G[s_i], s_j, l$);
18   UpdateNN($G[s_j], s_i, l$);
19          $c \leftarrow c + 1$;
20   **end**
21   **end**
22 **until** $c == 0$;
23 $U \leftarrow$ **Merge and Sort** G with G_1^v;
24 **return** *k*-NN Graph G

---

consistent performance. Without loss of generality, each *k*-NN list in G_1 is cut into equal halves by default. This setting remains valid as the scales of the two subsets are unbalanced.

Both S-Merge and J-Merge algorithms spend most of the costs on the distance computation (*Line 17* in Alg. 1 and *Line 16* in Alg. 2). The time costs that are spent on merge operation with the truncated graph (*Line 24* in Alg. 1 and *Line 23* in Alg. 2) are negligible compared to the cost on the 3rd step when the data dimensionality is high. Compared to S-Merge, J-Merge needs to perform cross-comparison between samples within the raw set. As a result, for a dataset of the same size, J-Merge requires more comparisons

than that of S-Merge.

3.3 Hierarchical *k*-NN graph Construction via J-Merge

Different from S-Merge and NN-Descent, J-Merge is capable of building an approximate *k*-NN graph for a dataset where its size grows incrementally. Specifically, the construction of an approximate *k*-NN graph starts when only a small subset of a dataset is available. By using J-Merge, the remaining subsets are incrementally combined with the increasingly growing *k*-NN graph. Therefore, J-Merge indeed can be applied as an incremental version of *k*-NN graph construction. The size of a subset to be joined in can be specified. Typically, when the size is equal to the current graph, a newly updated *k*-NN graph will double its size after merging.

Namely, the construction starts by building an approximate *k*-NN graph on a randomly sampled small subset from dataset S with NN-Descent. Thereafter, a raw subset of the same size as the baked one is sampled from the remaining dataset of S. J-Merge is called to join this subset into the approximate *k*-NN graph. After J-Merge, the *k*-NN graph grows two times bigger than before. In the next round of merge, the size of the block to be joined in is two times bigger than the previous round. This sampling and merging process is repeated for several rounds until all the samples in S are joined into the graph. The intermediate *k*-NN graphs produced during the whole iterations form a hierarchical (or pyramid) structure. The lower the layer is, the more samples are kept in the graph. The bottom layer is the approximate *k*-NN graph built for the whole set S. This approximate *k*-NN construction algorithm is called hierarchical merge (H-Merge). Essentially, H-Merge is a repetitive calling of J-Merge. It is important that the subset joined in the hierarchy each time should be randomly drawn from the whole set. This is to guarantee that different levels in the hierarchy reflect well the data distribution of different granularity.

Similar to the online *k*-NN graph construction algorithm [32], H-Merge deals with the *k*-NN graph construction problem for a close set as well as an open set. However, different from [32], the *k*-NN graph is constructed in a hierarchical manner. Compared to the hierarchical navigable small world (HNSW) graphs, H-Merge produces an approximate *k*-NN graph instead of a sparsified approximate *k*-NN graph in each layer. Furthermore, the hierarchy in H-Merge is formed by intermediate *k*-NN graphs during the merge. No real hierarchical structure is maintained during the construction. The approximate *k*-NN graph of one layer is derived from the approximate *k*-NN graph of the upper layer. The upper layer graph is merged into the approximate *k*-NN graph of the next layer after each round of J-Merge. During the construction process, one is allowed to save an arbitrary number of layers to form the hierarchy.

Additionally, similar to HNSW graphs, the hierarchical structure formed by H-Merge could be adopted for the nearest neighbor search. Like HNSW graphs, the links to the remote neighbors are kept in the upper layers of the hierarchy. These remote links help to skip a large number of far neighbors during the NN search. As a result, a higher speedup over NN search on a single flat graph is expected, particularly on low dimensional data. These remote links

also act as the bridges between clustered data such that the NN search escapes from being trapped in the clustered local. Furthermore, due to the high quality of the approximate *k*-NN graph, the NN search performance based on such hierarchy turns out to be even better than HNSW graphs in some cases, which will be revealed in the experiment section. In order to make better use of the hierarchical graphs produced by H-Merge, the graph in each hierarchy is undergone graph diversification before the hierarchical graphs are supplied to support NN search. This will be detailed in Section 4.

There is one parameter in the above merge algorithms, namely parameter k, which is also the size of the resulting *k*-NN graph. For S-Merge and J-Merge, k is specified as the user's wishes. In general, larger k leads to better *k*-NN graph quality while higher computation cost. Besides k, the *k*-NN graph quality is also impacted by the quality of the initial sub-*k*-NN graphs in both scenarios. In H-Merge, parameter k has a similar impact on the performance. Since the scale of approximate *k*-NN graphs of the non-bottom layer is small. It is no need to keep a full *k*-NN list for these graphs. As a result, on these layers, the size of the *k*-NN list is set to $\frac{k}{2}$.

Essentially, S-Merge, J-Merge, and H-Merge are the extensions of NN-Descent algorithm. S-Merge is designed for parallel approximate *k*-NN graph construction. J-Merge is designed to build an approximate *k*-NN graph dynamically. While H-Merge is proposed to support fast NN search for an open set. All the above three graph merge algorithms are generic to various distance metrics. The codes of all these algorithms have been open-sourced on GitHub[2].

3.4 Convergence, Optimality and Complexity Analysis

In this section, the convergence analysis is made for S-Merge. Since the optimization strategies used in S-Merge, J-Merge, and H-Merge are similar. The convergence analysis for S-Merge is also feasible for J-Merge, H-Merge as well as NN-Descent.

Given datasets $S_1 = \{s_i | s_i \in R^d\}$, $S_2 = \{s_j | s_j \in R^d\}$ and $S = S_1 \cup S_2$, U^0 is the union of augmented graphs G_1^u and G_2^u, which is the initial graph prepared for NN-Descent iteration. Given a *k*-NN graph G, we define function $\phi(G)$ that returns the sum of distances from all k neighbors of all samples kept in G.

$$\phi(G) = \sum_{i=1}^{n} \sum_{j=1}^{k} G_{ij}, \qquad (1)$$

where G_{ij} keeps the distance from the j-th neighbor to sample i. The set of the possible states for the approximate *k*-NN graph is finite and countable. Its size is $n \times C_n^k$, where n is the input dataset size and k is the size of each NN list. Essentially, NN-Descent simply jumps from one state (solution) to another more optimal one. Given the true *k*-NN graph for dataset S is $\mathcal{G}$, we have $\phi(U^0) > \phi(\mathcal{G})$ holds[3]. After one round iteration in Alg. 1, closer neighbors are joined into NN lists and far neighbors are swapped out. To make the update operation happens, there must exist a

---

2. https://github.com/wlzhao/nn-merge

3. Without the loss of generality, the shorter the distance is, the closer is the neighbor.

U_{ij}^0 being replaced by U_{ij}^1, where $U_{ij}^1 < U_{ij}^0$. So we have $\phi(U^0) > \phi(U^1)$. Given the series of intermediate k-NN graphs produced by the procedure are $U^1, U^2, \cdots, U^t, \cdots$, following inequation holds

$$\phi(U^0) > \phi(U^1) > \cdots \phi(U^t) > \phi(U^{t+1}) > \cdots \geq \phi(\mathcal{G}). \quad (2)$$

Notice that $U^1, U^2, \cdots, U^t, \cdots$ are only a few of all the possible states. Since the update on U^t happens only when a closer neighbor is found, the iteration leads function $\phi(U^t)$ to decrease monotonically. Meanwhile, the function value is lower-bounded by $\phi(\mathcal{G})$. It is therefore clear to see the iteration in S-Merge converges.

Both S-Merge and J-Merge are greedy optimization approaches. Similar to NN-Descent, S-Merge and J-Merge perform hill-climbing NN search [10] in a batch fashion. The greedy hill-climbing process could be trapped in local optima. While different from NN-Descent, S-Merge and J-Merge climb from the half-way up to the hill. Both S-Merge and J-Merge are effective on low dimensional data. As will be revealed later, their effectiveness drops as the data dimensionality increases.

The time complexity of NN-Descent is $O(d \cdot n^\rho)$, where $1.4 \leq \rho \leq 1.9$ according to our empirical study. In general, the cost of merging two graphs is lower than constructing the whole from scratch by NN-Descent. Given an n-scale dataset, we now have three ways to construct a k-NN graph by NN-Descent. The first way builds the graph by NN-Descent directly. In a second way, the construction is decomposed into three steps. The first two steps construct the sub-k-NN graphs for S_1 and S_2 by calling NN-Descent respectively, given $|S_1| = |S_2|$. In the 3rd step, S-Merge is called to merge two subgraphs. The number of comparisons in S-Merge is the number of comparisons of the first way excluding out the number of comparisons that happens between samples from the same subset. Since samples from two subsets have been mixed up sufficiently in the initialization stage, the chance that one sample in a k-NN list compares to a sample from the same subset is on the same level as the chance of being compared to a sample from another subset. Given $|S_1| = |S_2|$, the number of cross-comparisons between two subsets takes up one-third of the total comparisons. Namely, we have

$$P_s(n) = \frac{d \cdot n^\rho}{3}. \quad (3)$$

Since two subsets are already of high quality, the NN-Descent procedure converges faster than it runs on a completely raw set. The number of cross-comparisons between two subsets could be slightly smaller than $P_s(n)$. As a result, the time complexity upper bound of S-Merge is around one-third of NN-Descent.

In a third way, alternatively, J-Merge could be called to replace the 2nd and the 3rd steps in the second way. In this case, in addition to the number of cross-comparisons, the number of comparisons within the raw set should be counted. Since samples from two subsets have been mixed up sufficiently in the initialization stage, one sample in a k-NN list holds an equal chance to compare to a sample from the same subset and a sample from another subset. As a result, the number of comparisons in J-Merge takes up two-

TABLE 1: Time Complexity of S-Merge, J-Merge, and H-Merge. ρ is usually in the range of $[1.4, 1.9]$. It varies as the scale of the data and the type of the data change.

Algorithm	S-Merge	J-Merge	H-Merge	NN-Descent
Time Comp.	$O(\frac{d \cdot n^\rho}{3})$	$O(\frac{2 \cdot d \cdot n^\rho}{3})$	$O(\frac{5 \cdot d \cdot n^\rho}{3})$	$O(d \cdot n^\rho)$

thirds of the total comparisons in NN-Descent. So we have the time complexity of J-Merge as

$$P_j(n) = \frac{2 \cdot d \cdot n^\rho}{3}. \quad (4)$$

Since subgraph G_1 is already in good shape, the NN-Descent iteration procedure converges faster. The number of comparisons required in J-Merge is therefore slightly smaller than $\frac{2 \cdot n^\rho}{3}$.

In S-Merge and J-Merge, the extra time has to be spent on merge sort at the final step. Since each k-NN list is sorted, the merge sort is very efficient. The time complexity of this step is $O(n \cdot k)$. When both n and d are large, the time cost of this step is minor in contrast to the iteration procedure. Overall, the time complexity of S-Merge and J-Merge is around *30% ~70%* of NN-Descent.

Before we discuss the time complexity of H-Merge, let's consider the time complexity of building a series of approximate k-NN graphs with growing sizes by calling NN-Descent repetitively. Given the size of the graph is doubled each time, the scale of the problem is given as

$$s = \cdots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n, \quad (5)$$

where n is the size of the dataset. As a result, the overall scale of the problem is $2 \cdot n$. So the time complexity of building a hierarchy by the repetitive calling of NN-Descent is equivalent to building the whole graph twice by NN-Descent. Compared to this problem scale, H-Merge is more cost-saving since it performs J-Merge with an upper layer graph on each level. The scale of the problem is roughly two-thirds of NN-Descent of the same level as discussed before. As a consequence, the overall time complexity of H-Merge is roughly *1.67* times over the complexity of calling NN-Descent to build a flat graph. The time complexity of S-Merge, J-Merge, and H-Merge are summarized in Tab. 1.

4 NN SEARCH OVER *k*-NN GRAPH HIERARCHY

As demonstrated by HNSW graphs [13], the hierarchical graph structure is helpful for the NN search task. In this section, we are going to show that a similar hierarchical NN search mechanism could be undertaken with the graphs built by H-Merge.

As mentioned in the previous section, on each layer of the hierarchy, H-Merge produces an approximate k-NN graph. It is possible to conduct the top-down NN search directly based on the hierarchy formed by a series of approximate k-NN graphs. Similar to HNSW, the search starts from the top layer and performs hill-climbing on the graph until it reaches the closest sample on the layer. Thereafter the closest sample is taken as the starting point in the next layer search. The same search procedure is repeated on the
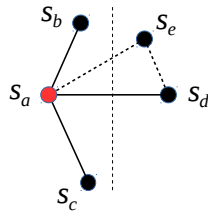
Fig. 2: The illustration of the heuristic strategy used to select the graph neighbors in each layer of the hierarchical structure. An edge from s_a to s_d occludes an edge from s_a to s_e since s_e is closer to s_d than s_a.

second layer with the supplied seed. This process continues until it finds the closest sample on the bottom layer.

Although the above search scheme is feasible, the comparison redundancy happens due to the occlusions in the k-NN neighborhood [13], [31], [35]. Following the practice in [13], [31], [35], the graph diversification is applied to the approximate k-NN graph of each layer. This leads to a sparse NN graph and therefore reduces the potential comparison redundancy. In this paper, the graph diversification (GD) operation scheme used in HNSW [13], [35] is adopted. However different from [13], the graph diversification is performed on the already built approximate k-NN graphs as a post-processing step. In contrast, HNSW applies the diversification on an incrementally diversified graph [13].

As shown in Alg. 3, GD examines the neighborhood of each k-NN list. Given sample s_a, the nearest neighbor of s_a is kept by default. The rest neighbors are treated as the candidates to be examined. The candidates are sorted in ascending order and are examined one by one. A candidate is kept if its distance to sample s_a is smaller than its distances to all the already kept samples. As illustrated in Fig. 2, sample s_e is removed from the NN list of sample s_a, since its distance to s_d is smaller than it is to s_a. The reverse k-NN list of sample s_a is also diversified based on the same rule and merged with the diversified k-NN list. The k-NN graph of each layer is undergone this diversification operation. The hierarchy after graph diversification is ready to support efficient NN search.

The NN search on the hierarchy largely follows the procedure of HNSW [13]. It is divided into two stages. At the first stage, a coarse greedy search is conducted on the non-bottom layers. The search starts from the top layer where we have the smallest graph. It starts from one random sample on the layer and explores the neighbors of a visited vertex. The closest neighbor in the expanded neighborhood is treated as the candidate to explore in the next round. The search stops when no closer neighbor to the query is found on this layer. The closest neighbor found at the upper layer is treated as the starting point of the search on the next layer. This process repeats until it reaches the last non-bottom layer. In the second stage, the discovered closest sample on the first phase is taken as the starting point for the bottom-layer search. Different from the non-bottom layer search, all the samples maintained in a top-ranked list are expanded during the search. The search visits the top-ranked list in a best-first fashion and terminates when no new sample in the rank list to be expanded. The search procedure on a three-
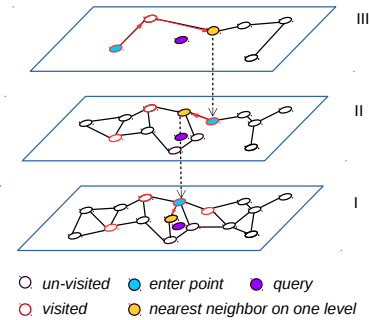

Fig. 3: The illustration of top-down NN search on a three-layer graph hierarchy.

layer hierarchy is illustrated in Fig. 3.

Similar to HNSW, the links (edges) kept in the upper layers connect vertices relatively far away from each other. Compared to the NN search on a single flat graph, the search moves faster when it is undertaken on these coarser graphs. Therefore the NN search on the non-bottom layers is expected to supply candidates to the bottom layer search in a more efficient way than search on a flat graph.

Algorithm 3: k-NN Graph Diversification

Data: $nhood$: neighborhood of sample s_a
Result: candidate set Q selected by heuristic

1   $Q \leftarrow nhood[1]$;
2   **for** *each $s_i \in nhood[2 \cdots k]$* **do**
3     $flag = true$;
4   **for** *each $c \in Q$* **do**
5   **if** $m(s_i, c) < m(s_i, s_a)$ **then**
6         $flag = false$;
7   break;
8   **end**
9   **end**
10   **if** $flag$ **then**
11      $Q \leftarrow Q \cup s_i$;
12   **end**
13   **end**
14   **return** Q

---

5 EXPERIMENTS

In this section, the performance of S-Merge and J-Merge is studied on the k-NN graph merge task. Each of the datasets is divided into two subsets. The sub-k-NN graphs are constructed in advance by NN-Descent. S-Merge and J-Merge are called respectively to fulfill the merge in two different scenarios. Since there is no graph merge algorithm in the literature, their performance is compared to NN-Descent, NSW [12] and RLB [7], each of which is adopted to construct the graph for the whole set directly. While H-Merge is evaluated when it is used as a hierarchical approximate k-NN graph construction approach. Meanwhile, the performance of H-Merge is also studied when the hierarchical structure is used to support fast NN search.

As revealed in [5], [31], the complexity of approximate k-NN graph construction is largely related to the intrinsic data

TABLE 2: Overview of Datasets.

Dataset	n	d	Qry	Exh. Srch (s)	LID [36]	$m(\cdot,\cdot)$	Type
RAND100K	1×10^5	$2 \sim 100$	-	-	-	l_2	synthetic
RAND100K	1×10^5	$2 \sim 100$	-	-	-	l_1	synthetic
SIFT100K [22]	1×10^5	128	-	-	-	l_2	SIFT [37]
MNIST [38]	7×10^4	784	-	-	-	l_2	Image
NYTimes [38]	3×10^5	256	-	-	-	$Cosine$	Text
Kosarak [38]	7.4×10^4	27,983	-	-	-	$Jaccard$	Itemset
RAND10M8D	1×10^7	8	1,000	78.8	6.5	l_2	synthetic
RAND10M32D	1×10^7	32	1,000	26.1	19.4	l_2	synthetic
RAND1M100D	1×10^6	100	1,000	92.0	48.9	l_2	synthetic
NUSWide	2.6×10^5	500	1,000	958.1	38.2	κ^2	BoVW [39]
SIFT1M [22]	1×10^6	128	10,000	1184.9	16.3	l_2	SIFT [37]
SIFT10M [22]	1×10^7	128	10,000	11441.0	16.3	l_2	SIFT [37]
GIST1M [40]	1×10^6	960	1,000	919.1	38.1	l_2	GIST [40]
GloVe1M [41]	1.2×10^6	100	1,000	109.3	39.5	$Cosine$	Text
YFCC1M [42]	1×10^6	128	10,000	1091.6	23.4	l_2	Deep Feat.
Kosarak [38]	7.4×10^4	27,983	500	66.5	47.3	$Jaccard$	Itemset

dimension. The intrinsic data dimension of real-world data varies considerably from one data type to another. Different from real-world data, the intrinsic dimension of synthetic data increases steadily as the data dimension increases. As a result, the performance trend of *k*-NN merge approach on synthetic data is more observable compared to the real-world data. For this reason, the experiments of *k*-NN graph merge are conducted on a series of synthetic datasets. The data dimension varies from *2* to *100*, which is in line with the convention in [5]. Values in each dimension are independently drawn from the range $[0, 1)$ under the uniform distribution. Moreover, another four real-world datasets are further adopted to demonstrate the performance of S-Merge and J-Merge on different data types. Datasets MNIST and Kosarak are composed of sparse vectors. The densities of these two datasets are *0.198%* and *19.270%* respectively.

For the NN search task, the performance is reported on both synthetic random data and data from the real-world. Besides three large-scale synthetic datasets, seven real-world datasets are adopted in the evaluation. The brief information about all the datasets is summarized in Tab. 2. Among them, seven datasets are derived from real-world images, deep features, or text data. Four datasets, namely GIST1M, GloVe1M, NUSWide, and Rand1M that are marked as most challenging datasets in [31], are incorporated. For each of the datasets, another set of queries of the same data type are prepared. Different metrics such as l_2, *Cosine*, κ^2, and *Jaccard* are adopted in accordance with the data type of each set.

On the NN search task, the performance of the proposed search approach is studied in comparison to the representative approaches of different categories. Namely, they are graph-based approaches such as DPG [31], SPTAG[4] [43] and HNSW [13]. SRS [44] is considered as the representative locality-sensitive hash approach. Product quantizer (PQ) [22] is considered as the representative quantization based approach in the comparison. FLANN [19] and Annoy [45] are selected as the representative tree partitioning approaches, both of which are popular NN search libraries in the literature.

---

4. In favor of search quality, SPTAG-BKT is used in the experiment.

5.1 Evaluation Protocol

For *k*-NN graph construction, the top-*1* (*recall@1*) and top-*10* (*recall@10*) recalls on each dataset are studied under l_1 and l_2 metrics respectively. Given function $R(i, k)$ returns the number of truth-positive neighbors at top-*k* NN list of sample i, the recall at top-*k* on the whole set is given as

$$recall@k = \frac{\sum_{i=1}^{n} R(i,k)}{n \times k}. \qquad (6)$$

Besides *k*-NN graph quality, the construction cost is also studied by measuring the scanning rate [5] of each approach. Given C is the total number of distance computations in the construction, the scanning rate is defined as

$$c = \frac{C}{n \times (n-1)/2}. \qquad (7)$$

The search quality is measured by the top-*1* recall for the first nearest neighbor. This is in line with the evaluation convention in the literature. Moreover, in order to allow the readers to know how efficient that one NN search approach performs, the time cost for brute-force NN search on each dataset is also shown on the *5*th column of Tab. 2.

All the codes of different approaches considered in this study are compiled by g++ *5.4*. The multi-threads, SIMD, and pre-fetching instructions are enabled as long as they are supported in the source codes for NN search task. While for *k*-NN graph merge task, OpenMP [46] is adopted. This is to illustrate our merge algorithms are parallelizable on the CPU threads level as NN-Descent. All the experiments are pulled out on a workstation with *24* cores of *2.4*GHz CPUs and *64*G memory setup.

5.2 Performance of S-Merge and J-Merge

In this section, the performance of two proposed merge algorithms is studied in comparison to NN-Descent [5], which is recognized as the state-of-the-art approximate *k*-NN graph construction algorithm. The evaluation is conducted on six synthetic datasets. While the size of the datasets is fixed to *100*K, which is in line with [5]. The performance is reported when the merge algorithms are operated under l_1 and l_2 metrics respectively. The graph quality achieved by S-Merge, J-Merge, and NN-Descent is largely controlled by *k*, which is exactly the length of the
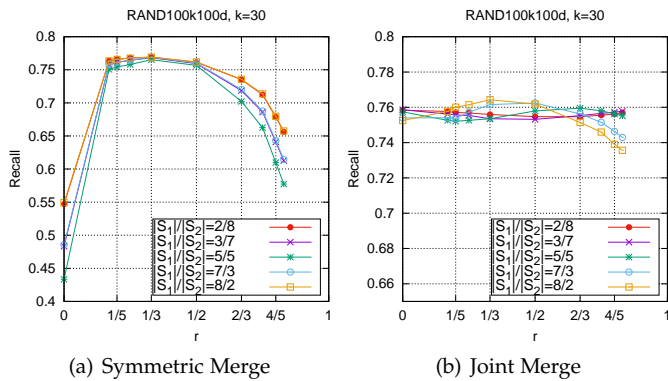
Fig. 4: The fluctuation trend of graph quality when varying parameter r in two merge strategies on RAND100K100D dataset. Parameter k is fixed to 30. Figure (a) and Figure (b) show the variation of recall@10 when parameter r changes from 0 to $4/5$ for S-Merge and J-Merge respectively. In both cases, the size ratio of between S_1 and S_2 is varied from $2/8$ to $8/2$. To ensure the statistical significance, the digits are reported by averaging 20 rounds of results for each parameter setting.

NN list. In our experiment, k is set to the value where the algorithm makes a balance between efficiency and quality.

Each dataset is divided into two subsets. For S-Merge, NN-Descent is called to build two sub-k-NN graphs respectively for two subsets. For J-Merge, NN-Descent is called to build the k-NN graph for one of the subsets, while leaving another as the raw set. It is possible to use approach [32] to produce the subgraph, which actually achieves better quality. However, NN-Descent is preferred in the test as both S-Merge and J-Merge can be viewed as the extensions over NN-Descent. It is easy to see the efficiency and quality that S-Merge and J-Merge achieve over NN-Descent.

In S-Merge and J-Merge, the number of samples in each NN list that is cut out for the final merge could range from 0 to k. Given the number of samples reserved for mixture-up is k_1, the ratio that regularizes this division is defined as $r = \frac{k_1}{k}$. This factor impacts the performance of both S-Merge and J-Merge. In the following, an ablation analysis is made to see how this factor impacts the performance of the graph merge in two scenarios.

5.2.1 Ablation Analysis

The ablation analysis about parameter r is conducted on "RAND100K100D" with l_2 distance measure. k is set to 30. As mentioned before, NN-Descent is called to produce subgraphs. Then S-Merge and J-Merge are called to merge the approximate sub-k-NN graphs. The size ratio between two subsets, namely $\frac{|S_1|}{|S_2|}$ is varied from $\frac{2}{8}$ to $\frac{8}{2}$. So that both the balanced and unbalanced cases of graph merge are studied. Under each size ratio, different settings for parameter r are tested.

Fig. 4(a) and Fig. 4(b) show the curves of the graph quality variation under each size ratio when r varies from 0 to $\frac{4}{5}$ for S-Merge and J-Merge respectively. As shown in Fig. 4(a), the graph quality from S-Merge is maintained on the high level only when r is in the range $[\frac{1}{5}, \frac{1}{2}]$. Within this

range, it allows the samples from two sides to be exposed to the cross-comparison sufficiently. In contrast, J-Merge is less sensitive to r. As shown in Fig. 4(b), in the case $|S_1| \leq |S_2|$, the graph quality varies a little even when no samples from S_2 are joined into the k-NN list of subgraph G_1. This is because samples from S_1 and S_2 are already present in the initial graph G_2. It already guarantees that the samples from both sides are sufficiently exposed to cross-comparison. In the case $|S_1| > |S_2|$, the initial graph G_2 is dominated by samples from S_1. The raw samples have to be joined into the k-NN list of G_1 to be better exposed to the cross-comparison. As shown in Fig. 4(b), a valid range for r in J-Merge is $[\frac{1}{3}, \frac{1}{2}]$. In the meantime, the higher chance of being exposed to cross-comparison induces more number of distance computations. This in turn leads to the higher computation cost as well as the higher graph quality. In practice, a valid setting for r is $\frac{1}{2}$ in both merge algorithms as we want to maintain similar graph quality as NN-Descent.

5.2.2 Performance on approximate k-NN graph Merge

In this section, the effectiveness, as well as the efficiency of the two merge algorithms, are studied on both the synthetic and real-world datasets. For convenience, the performance is evaluated in the scenario of merging two subsets with equal size. In the test for S-Merge, two sub approximate k-NN graphs are prepared by NN-Descent in advance. S-Merge is called to merge the two. In the test for J-Merge, one sub approximate k-NN graph is prepared by NN-Descent. J-Merge is called to merge the left raw set into the subgraph. Their performance is compared to incremental NN-Descent (given as NN-Descent*), online k-NN graph construction approach NSW [12], and divide-and-conquer k-NN graph construction approach RLB [7]. For NN-Descent*, the first half k-NN graph is constructed by NN-Descent. The rest of the samples are inserted into the k-NN graph one by one by the NN search procedure provided in the same package [5]. NN-Descent* and J-Merge functionally overlap with each other. Since no mechanism is designed in NSW to merge two existing graphs, samples from one sub-graph are inserted into another by NSW algorithm one by one. In another word, both joint merge and symmetric merge are treated as joint merge in NSW. The parameter R of the RLB code[5] is tuned (from 0.07 to 0.3) with respect to different data dimensions. Parameter k in RLB is fixed to 10, the setting of which is a trade-off between graph quality and time cost. The performance from NN-Descent is still presented when it is called to produce an approximate k-NN graph directly for the whole set. This is to show how much cost the graph-merging algorithm could save compared to building the graph from scratch.

For different datasets, the parameter k for NN-Descent, NN-Descent*, NSW and our merge algorithms varies. It is largely close to the data dimension. While k is shared the same across different approaches on the same dataset. The first evaluation is conducted on six synthetic datasets and four real-world datasets. Top-1 and top-10 recalls for all five approaches are shown in Fig. 5. Correspondingly, the parameter setting for k, the scanning rate, and the wall-time for all the runs are presented in Tab. 3.

5. Codes are collected from https://jiechenjiechen.github.io.

TABLE 3: Construction scanning rate (c) and wall-time (tm) in seconds (s) of S-Merge and J-Merge in comparison to NN-Descent, NN-Descent*, NSW, and RLB on six synthetic and four real-world datasets. The dimension (d) of these synthetic datasets ranges from 2 to 100. For S-Merge and J-Merge, $|S_1| = |S_2|$. All the approaches are tested on a single core.

Dataset			NN-Desc.		NN-Desc.*		NSW		S-Merge		J-Merge		RLB	
$m(\cdot)$	d	k	c	tm	c	tm	c	tm	c	tm	c	tm	c	tm
l_1	2	10	0.009	**2.611**	0.003	6.200	0.006	3.031	0.006	3.996	0.006	2.755	**0.001**	7.734
	5	10	0.010	2.995	0.003	6.734	0.006	**2.457**	0.004	2.988	0.007	3.169	**0.003**	19.334
	10	20	0.038	8.656	0.006	19.111	0.019	6.247	0.012	**5.570**	0.021	7.796	**0.004**	32.862
	20	20	0.048	12.836	0.014	19.408	0.028	9.216	0.014	**7.827**	0.028	10.748	**0.009**	114.113
	50	30	0.120	42.124	0.052	83.090	0.075	34.268	**0.034**	**18.444**	0.071	30.786	0.066	1475.360
	100	40	0.208	120.066	0.087	185.087	0.127	99.205	**0.059**	**42.394**	0.121	79.437	0.141	5081.260
l_2	2	10	0.008	**2.673**	0.003	6.004	0.006	2.928	0.006	4.061	0.006	2.763	**0.001**	7.183
	5	10	0.010	3.122	0.003	6.052	0.005	**2.221**	0.004	2.979	0.007	3.090	**0.002**	18.374
	10	20	0.036	7.559	0.006	10.126	0.019	5.894	0.013	**5.759**	0.022	7.787	**0.003**	31.502
	20	20	0.051	13.264	0.014	19.671	0.028	9.186	0.015	**8.093**	0.030	11.092	**0.007**	116.916
	50	30	0.129	44.398	0.055	88.129	0.079	38.992	**0.038**	**19.859**	0.077	32.378	0.067	1469.280
	100	40	0.216	126.386	0.079	175.916	0.150	139.409	**0.064**	**45.972**	0.126	83.926	0.141	5093.500
		k	c	tm	c	tm	c	tm	c	tm	c	tm	c	tm
SIFT100K		20	0.056	45.019	0.022	66.078	0.029	26.551	0.019	**19.426**	0.035	31.074	**0.004**	53.918
MNIST		40	0.191	368.498	0.017	**71.542**	0.242	254.647	0.067	136.504	0.115	227.231	**0.012**	379.126
NYTimes		40	0.065	836.950	0.026	1020.780	0.033	665.268	0.021	**313.516**	0.038	528.336	**0.012**	4091.490
Kosarak		40	0.243	963.505	0.154	1140.600	-	-	**0.088**	**343.802**	0.150	557.900	-	-

As shown in the figure, the graph quality achieved by the two merge algorithms is largely similar to NN-Descent on both synthetic and real-world datasets. On the synthetic datasets, the graph quality that is produced by NN-Descent is slightly higher when the data dimension is in the range of $[10, 100]$. However, the performance difference between them is within 3%. On the other hand, the scanning rates of S-Merge and J-Merge are below the theoretical baselines that we discussed in Section 3.4. NN-Descent shows the highest scanning rate. The scanning rates from J-Merge are always higher than that of S-Merge across all the runs. However, according to our observation, S-Merge actually shows slightly higher scanning rates than J-Merge if the number of comparisons spent on building subgraphs is also considered. In the meantime, J-Merge shows slightly better performance than S-Merge. This trend is subtle but consistent. This does indicate it is more cost-effective by calling J-Merge than calling NN-Descent followed by S-Merge to merge a raw set. When a raw set is joined into a built graph, the raw samples are moving along a structured graph, which is easier for them to find true close neighbors. Compared to J-Merge, NN-Descent* shows lower scanning rate while considerably higher time costs. This trend is consistent except for MNIST, which is in low intrinsic dimension and small scale. In NN-Descent*, in addition to distance computation, extra cost is induced to maintain a dynamic reverse k-NN graph. Meanwhile, J-Merge outperforms NN-Descent* particularly on high dimensional datasets. The former is more cost-effective as it performs the hill-climbing in batch.

As shown in Fig. 5, both NSW and RLB show considerably poorer graph quality than the other approaches across all the datasets. While as shown in Tab. 3, the computation cost of NSW is close to J-Merge when the data dimension is low. As the data dimension increases, its computation cost becomes much higher than J-Merge. It is clear to see that fast online k-NN graph construction is not the appropriate solution for the graph merge problem. The major difference between J-Merge and NSW is that J-Merge performs "hill-climbing" batchfully. RLB only performs relatively well on high-dimensional data with low intrinsic dimension such as MNIST. NSW and RLB perform much poorer than NN-Descent and our merge algorithms. Comparing Fig. 5(a)-(b) with Fig. 5(a)-(b), the performance varies across different distance metrics for the data on the same dimension. The data are distributed differently under different metric spaces. k-NN graph construction for data in l_1-space is harder than that of l_2-space, where the data samples are easier to reach to each other via neighborhood links.

As seen from the performance reported on the real-world datasets (bottom table in Tab. 3), most of the approaches show considerably higher scanning rates on sparse datasets MNIST and Kosarak than that of SIFT100K and NYTimes. This does indicate the greedy graph-based approaches are not the best option for sparse itemsets [11].

In terms of running time, the costs from S-Merge and J-Merge are close to that of NN-Descent when the data dimension is low. As shown in Tab. 3, the time costs from S-Merge and J-Merge are even slightly higher than NN-Descent for datasets of 2-dimension and 5-dimension. In these cases, the time costs spent on distance computation are close to the cost in the merge sort step, which causes extra computation overhead. Nevertheless, as the data dimension grows, the distance computation dominates the overall cost, while the time cost for merge sort becomes minor. As the data dimension is higher than 20, the time costs from S-Merge and J-Merge are around one-third and two-thirds of NN-Descent, which meets well with the analysis presented in Section 3.4.

The efficiency of S-Merge and J-Merge is further studied on the multi-threads context. Similar to NN-Descent, both S-Merge and J-Merge are parallelizable on the CPU threads level. The experiment is pulled out on two large-scale datasets SIFT1M and GIST1M. Each dataset is divided into two subsets with equal size. Following the same pipeline as before, S-Merge and J-Merge run on multiple cores to merge the subgraphs. NN-Descent is treated as the comparison baseline.

Fig. 6 shows the wall-time of three algorithms as the number of cores increases incrementally. As shown in the figure, all algorithms show steady speedup when the num-

TABLE 4: Performance on *k*-NN graph Construction for *10* large-scale datasets. All the approaches run on *24* cores. HNSW and SPTAG are not designed for *k*-NN graph construction. The *k*-NN graph quality from them is listed for reference only.

Dataset	RAND10M8D		RAND10M32D		RAND1M100D		NUSWide		SIFT1M	
	Time (s)	Recall@10	Time (s)	Recall@10	Time (s)	Recall@10	Time (s)	Recall@10	Time (s)	Recall@10
H-Merge	597.07	**1.000**	135.74	0.985	288.88	0.532	1372.52	**0.869**	251.45	**0.998**
HNSW	205.36	-	394.74	0.740	580.38	0.254	190.37	0.151	187.23	0.532
NN-Descent	**132.84**	0.998	**56.65**	**0.987**	**154.38**	**0.544**	**757.68**	0.849	**136.07**	0.997
SPTAG	6132.00	0.294	2123.00	0.613	4555.00	0.479	-	-	2490.00	0.478

Dataset	YFCC		GIST1M		GloVe1M		Kosarak		SIFT10M	
	Time (s)	Recall@10	Time (s)	Recall@10	Time (s)	Recall@10	Time (s)	Recall@10	Time (s)	Recall@10
H-Merge	264.90	**0.980**	1345.75	**0.969**	369.37	**0.921**	117.724	0.967	3549.86	**0.995**
HNSW	252.48	0.600	**591.28**	0.252	304.82	0.462	-	-	2458.55	0.588
NN-Descent	**133.91**	0.975	990.98	0.966	**155.70**	0.913	76.677	**1.000**	**1620.41**	0.993
SPTAG	3338.00	0.605	17352.00	0.224	3590.00	0.293	-	-	35150.00	0.604

(a) Recall@1 with l_1-norm

(b) Recall@10 with l_1-norm

(c) Recall@1 with l_2-norm

(d) Recall@10 with l_2-norm
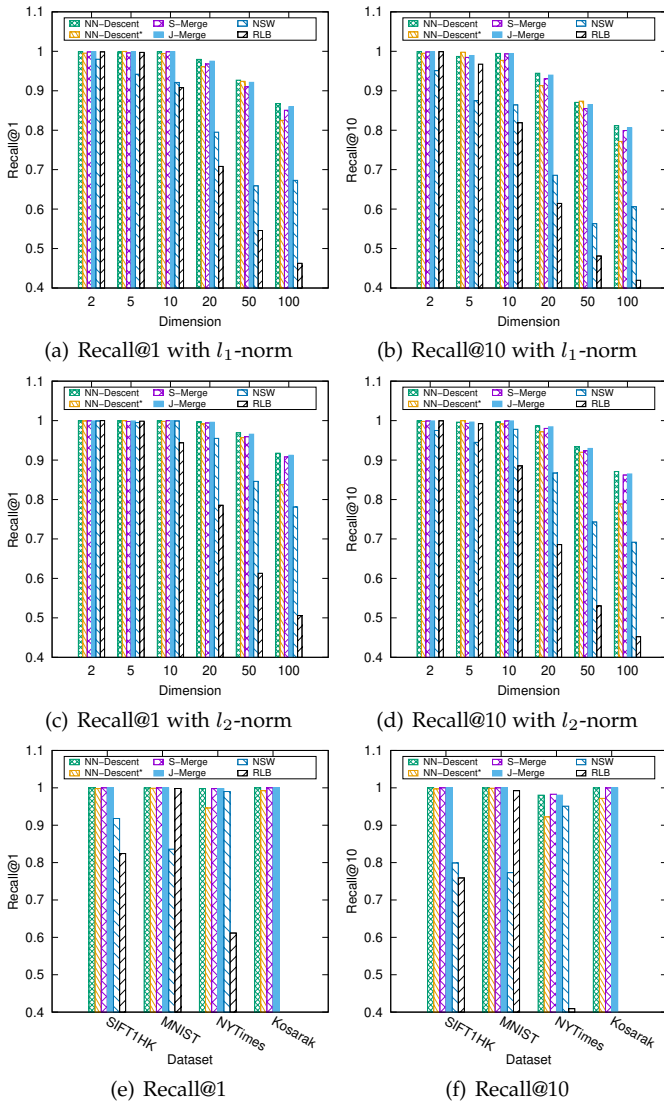
(e) Recall@1

(f) Recall@10

Fig. 5: The quality of approximate *k*-NN graphs produced by S-Merge and J-Merge, in comparison to NN-Descent, NSW and RLB on six synthetic datasets and four real-world datasets. On the synthetic datasets, both l_1-norm and l_2-norm are tested. On all the runs for S-Merge and J-Merge, $|S_1| = |S_2|$.
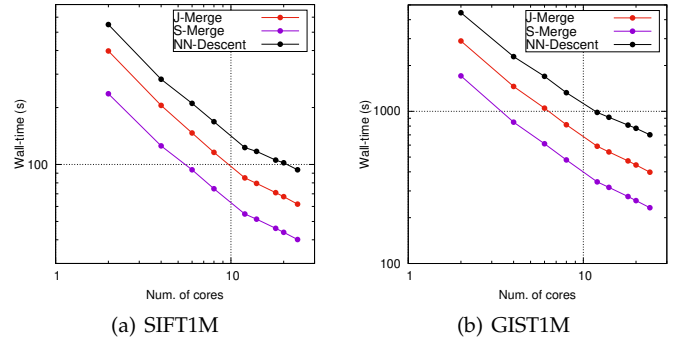

(a) SIFT1M

(b) GIST1M

Fig. 6: The efficiency of S-Merge and J-Merge in comparison to NN-Descent when they run on multiple threads. The experiment is conducted by increasing the number of cores incrementally from *2* to *24*. *k* is fixed to *30* on both datasets.

ber of cores increases. The speedup becomes less significant when the number of cores reaches *12*.

S-Merge and J-Merge are essentially the extensions over NN-Descent. Both of them maintain similar efficiency and effectiveness of NN-Descent. Although their overall costs and performance are similar, they play different roles in the approximate *k*-NN graph construction. S-Merge is suitable for parallel or distributed approximate *k*-NN graph construction. J-Merge allows an approximate *k*-NN graph to be incrementally built. While the original NN-Descent is suitable for building a graph for a fixed dataset or build subgraphs for S-Merge and J-Merge use.

5.3 Performance on NN Search

In this section, we study the performance of H-Merge as it is adopted to build the hierarchy graphs for the fast NN search. The search follows the pipeline described in Section 4. In the hierarchy, five layers are kept for datasets except for Kosarak. The sizes of graphs in each layer (from top to bottom) are set to *64, 512, 4,096, 32,768* and *n*, where *n* is the size of the whole reference set. For Kosarak, only four layers are kept. This setting is fixed all the time in the experiments. Ten datasets are used in the evaluation. They are one *10* million, two *1* million level synthetic datasets, and seven real-world datasets. The data dimension ranges from *4* to *27,983*. The reason to incorporate a series of synthetic datasets is to make the NN search performance trend more observable because their intrinsic data dimension increases
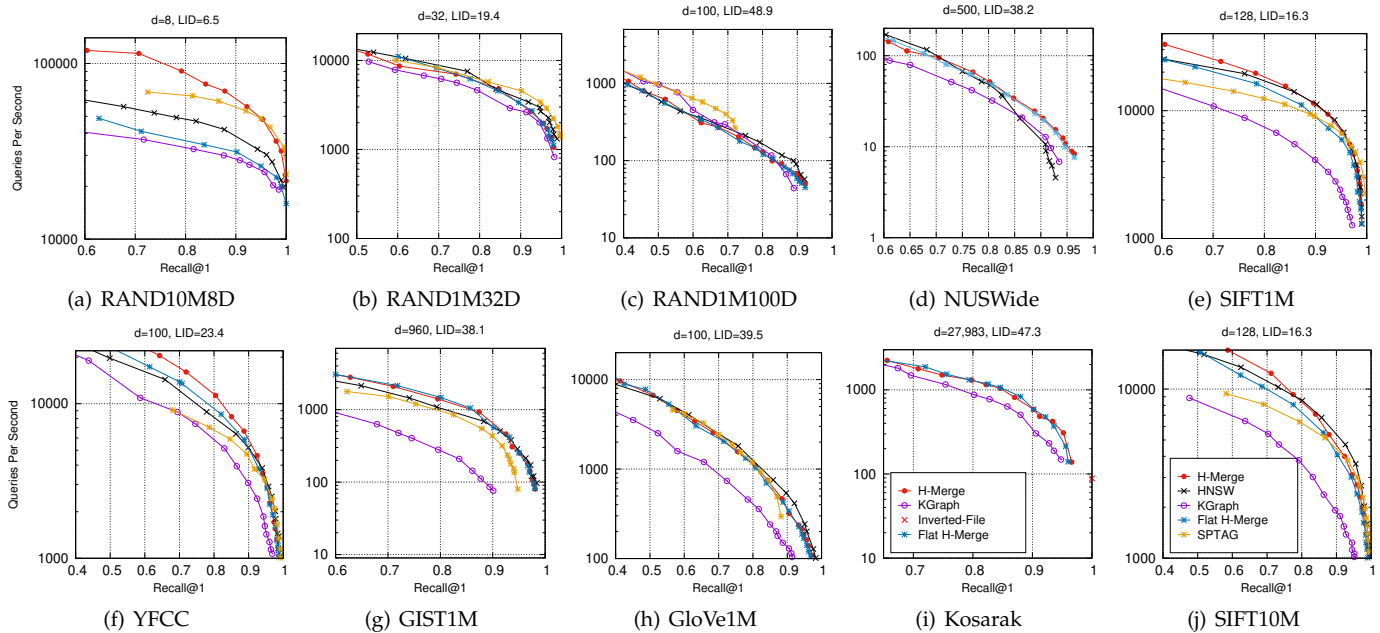
Fig. 7: The NN search performance from representative graph-based approaches on ten datasets. The performance is given as the number of queries one approach could process in one second against Recall@1 on this speed level. The series of #queries/recall pairs for each curve are produced by varying the critical parameter k in the search. Parameter k is the size of the top-rank list in the search procedure. Generally, larger k leads to higher search quality while lower speed efficiency.

steadily with respect to their data dimension. The general information is seen in Tab. 2. For all of the datasets except GloVe1M, NUSWide, and Kosarak, l_2 distance measure is used. For GloVe1M, we use *Cosine* distance. κ^2 distance and *Jaccard* distance are used for NUSWide and Kosarak respectively. In the study, we compare the search performance of H-Merge to recent graph-based approaches as well as the representative approaches of other categories.

5.3.1 Comparison to Graph-based Approaches

In the first evaluation, the performance of H-Merge is compared to other state-of-the-art graph-based approaches, namely HNSW [13], SPTAG [43], and KGraph [5]. For KGraph, the approximate k-NN graph is built by NN-Descent. An enhanced hill-climbing [10] procedure is adopted to perform the NN search over the approximate k-NN graph. Each k-NN graph in the graph hierarchy produced by H-Merge is diversified by Alg. 3. In addition, in order to investigate the impact of hierarchical structure in H-Merge, another run for H-Merge is undertaken. In this run, the bottom layer graph that is produced by H-Merge and diversified by Alg. 3 is directly used for the NN search. The search procedure is basically the same as the second phase search (described in Section 4) except that the seeds are randomly selected. This run is given as "Flat H-Merge". On datasets NUSWide and Kosarak, the performance from SPTAG is not reported because κ^2 distance and *Jaccard* distance are infeasible for SPTAG. Instead, the performance from Inverted-File is reported on Kosarak as it is believed to be effective for the high dimensional sparse datasets.

For H-Merge and KGraph, the hyper-parameter k is fixed to *40* across all the datasets. On the one hand, this setting is recommended in [31]. On the other hand, our

offline test confirms that this setting makes a good trade-off between the search quality and efficiency. In HNSW, parameters M and *efConstruction* are set to *20* and *128* respectively. Parameter M is the allowed connections to each sample, which is comparable to the allowed neighbors in a diversified graph [31]. Since the allowed neighbors in our diversified H-Merge bottom graph are set to $k/2$ (*i.e.,* 20), the parameter settings in H-Merge and HNSW are comparable. An offline test on M and *efConstruction* with different settings also confirm that current settings are appropriate across different datasets. In SPTAG, most of the parameters except for *TPTNumber* follow the default settings. Parameter *TPTNumber* is set to *8*, which improves search performance on most datasets while reducing index construction time.

The graph quality as well as the computation cost to build the approximate k-NN graph for *10* datasets are shown in Tab. 4. For all the approaches studied here, parameter k is fixed to *40*. For the all approaches, the graph construction is sped up by OpenMP on *24* cores. In general, H-Merge takes roughly twice more time to produce the approximate k-NN graph than NN-Descent. This is a little bit above the complexity bound we analyzed in Section 3.4 due to the computation overhead induced by additional operations. However, this extra cost brings us an extra bonus, namely the graph hierarchy. The graph quality of H-Merge and NN-Descent is similar in most of the cases. For HNSW and SP-TAG, many true neighbors have been deliberately removed from the graph, their approximate k-NN graph quality is therefore considerably lower than the other. Both HNSW graphs and SPTAG are primarily designed to support NN search instead of k-NN graph construction.

The search performance of four approaches is shown in Fig. 7. The performance is given as the number of queries

that one approach could process in one second against the Recall@1 on this speed level. It is clear to see H-Merge, HNSW, and Flat H-Merge outperform KGraph by a large margin on most of the datasets. Since the quality of graphs that support the NN search for KGraph and Flat H-Merge is similar, the performance gap between them largely owes to the graph diversification operation that is adopted in Flat H-Merge. H-Merge and HNSW show significantly superior performance over Flat H-Merge on the low dimensional datasets such as RAND10M8D due to the support of the hierarchy structure. This is the unique advantage of H-Merge and HNSW over other graph-based approaches such as [5], [31], [34]. The performance of SPTAG is poorer than H-Merge and HNSW on most of the datasets.

However, the performance superiority achieved by these two approaches fades away as the data dimension rises up to *32*. As the dimensionality grows, both hierarchical and non-hierarchical approaches get more likely trapped in the close neighborhood. The speedup that one approach could achieve is related to the intrinsic dimensionality of the dataset. The high speedup is achieved when the ratio between the data dimension and the intrinsic data dimension is high. This observation is in line with the findings in [31], [47].

Among the two approaches that show the best performance, H-Merge outperforms HNSW considerably on datasets RAND10M8D, YFCC1M, NUSWide, and GIST1M while shows similar performance as HNSW on the rest. Compared to HNSW, the advantage of H-Merge is that it applies graph diversification on a complete approximate *k*-NN graph. The structure in the NN list neighborhood is well-preserved. H-Merge is more attractive over HNSW for at least two reasons. First of all, H-Merge shows relatively better performance on various data types. Moreover, H-Merge is able to produce and maintain a high quality approximate *k*-NN graph for an open set. It is particularly helpful for multimedia websites, where we should maintain a dynamic *k*-NN graph to allow users to browse over similar photos and videos and support fast NN search in the meantime.

On the high dimensional sparse dataset (shown in Fig. 7(i)), the performance from H-Merge and Flat H-Merge is still competitive with Inverted-File, which attains *100%* recall while with slightly lower speedup. We believe that Inverted-File will show more significant superiority over graph-based approaches when the data dimensionality reaches millions.

5.3.2 Comparison to Approaches in the Literature

In this evaluation, we further study the performance of H-Merge in comparison to NN search approaches in the literature. Eight representative approaches are considered in the comparison. Namely, they are SRS [44], PQ [22], FLANN [19], Annoy [45], DPG [31], KGraph [5] and HNSW [13]. For DPG, its indexing graph is derived from the approximate *k*-NN graph produced by NN-Descent. So it shares the same approximate *k*-NN graph as KGraph. For all the approaches considered here, the parameters are set according to either the original paper or codes released by the authors. Four datasets that range from "easy" to "hard" are selected in the comparison. For each approach, we report
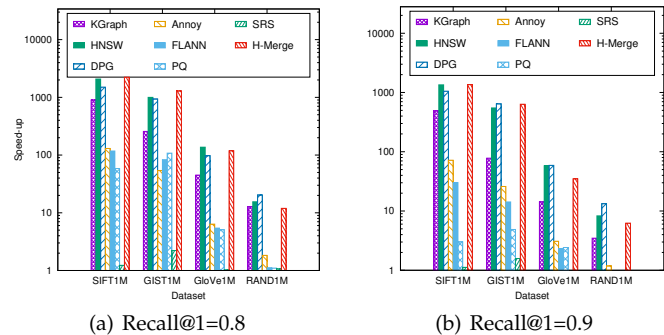

(a) Recall@1=0.8 (b) Recall@1=0.9

Fig. 8: The performance of H-Merge in comparison to representative approaches in the literature on four million level datasets. In the evaluation, we study the speedup that one approach could achieve when the search quality is fixed to Recall@1=0.8 and Recall@1=0.9 respectively. The speedup is measured in contrast to the exhaustive search time cost shown in Tab. 2.

its speedup over brute-force search when the top-*1* search recall level is fixed at *0.8* and *0.9* respectively. Fig. 8 (a) and Fig. 8 (b) show the speedups on these two different levels.

As seen from the figure, graph-based approaches H-Merge, DPG, and HNSW outperform the approaches of other categories considerably. Compared to non-hierarchical graph-based approaches (DPG and KGraph), the performance from H-Merge and HNSW is overall slightly better. However, no significant performance gap is observed between hierarchical approaches (e.g., HNSW and H-Merge) and non-hierarchical approaches (e.g., DPG and KGraph) on these high dimensional datasets. This again confirms that the hierarchy structure is able to slightly alleviate the complexity induced by the increase of data dimension but unable to overcome it. The hierarchy structure is helpful when both the data dimension and the intrinsic data dimension are low. In general, HNSW and H-Merge perform similarly on these two snapshot of views.

In this paper, three graph merge algorithms are proposed. The theoretical convergence analysis of these algorithms (including NN-Descent) is also given. Although S-Merge, J-Merge, and H-Merge are built on top of NN-Descent, the problems that S-Merge, J-Merge, and H-Merge aim to address are new. NN-Descent alone is unable to solve the problems. They play different roles in the approximate *k*-NN graph construction. S-Merge is suitable for parallel or distributed approximate *k*-NN graph construction. J-Merge allows an approximate *k*-NN graph to be incrementally built. From the practical point of view, we need all these approaches to be in place given the reminiscence of the big data problem from various fields. The original NN-Descent is only suitable for building a graph for a fixed dataset or building subgraphs for S-Merge and J-Merge use.

In addition to *k*-NN graph construction, H-Merge is proposed to build a hierarchical indexing structure for fast NN search. It shows competitive performance with state-of-the-art approach HNSW. Compared to HNSW, a *k*-NN graph is maintained. On the one hand, it facilitates the users to browse over NN search results via the links between

neighbors in the task such as image/video infringement search. On the other hand, it is also the media that allows two indexing structures to be merged via S-Merge. In contrast, such functionalities are not available in HNSW.

6 CONCLUSION

We have addressed the issues of merging *k*-NN graphs, which have been long overlooked in the literature. Three simple but effective solutions namely S-Merge, J-Merge, and H-Merge are proposed. They can be viewed as extensions over the classic NN-Descent algorithm and are tailored to addressing the issues that are unsolvable with the original NN-Descent. S-Merge is designed to merge two existing graphs, which is the critical step for parallel approximate *k*-NN graph construction. J-Merge addresses the problem of building an approximate *k*-NN graph for an open set, which is hardly achievable with state-of-the-art approaches. Deriving from J-Merge, H-Merge builds the approximate *k*-NN graph in a hierarchical manner, which facilitates fast NN search, particularly for low dimensional data. Moreover, the hierarchical graphs built by H-Merge could grow incrementally as well with the support of J-Merge. All the merge algorithms presented in the paper maintain the coherence, genericness, and simplicity exhibited in NN-Descent. Extensive experiments have been carried out to verify the effectiveness of the proposed solutions both for approximate *k*-NN graph construction and large-scale NN search.

REFERENCES

[1] J. B. Tenenbaum, V. de Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 5500, pp. 2319–2323, Dec. 2000.

[2] S. T. Roweis and L. K. Saul, "Report nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 5500, pp. 2323–2326, Dec. 2000.

[3] M. Wang, W. Fu, X. He, S. Hao, and X. Wu, "A survey on large-scale machine learning," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2020.

[4] J. Wang and Y. Xia, "Fast graph construction using auction algorithm," in *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, 2012.

[5] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th International Conference on World Wide Web*, WWW'11, (New York, NY, USA), pp. 577–586, ACM, 2011.

[6] C. Fu and D. Cai, "EFANNA : An extremely fast approximate nearest neighbor search algorithm based on knn graph," *arXiv.org*, 2016. arXiv:1609.07228.

[7] J. Chen, H. ren Fang, and Yousef, "Fast approximate knn graph construction for high dimensional data via recursive lanczos bisection," *Journal of Machine Learning Research*, vol. 10, pp. 1989–2012, Dec. 2009.

[8] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li, "Scalable k-nn graph construction for visual descriptors," in *CVPR*, pp. 1106–1113, Jun. 2012.

[9] Y.-M. Zhang, K. Huang, G. Geng, and C.-L. Liu, "Fast knn graph construction with locality sensitive hashing," in *Proceedings of the 2013th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part II*, ECMLPKDD'13, pp. 660–674, Springer-Verlag, 2013.

[10] K. Hajebi, Y. Abbasi-Yadkor, H. Shahbazi, and H. Zhang, "Fast approximate nearest-neighbor search with k-nearest neighbor graph," in *International Joint Conference on Artificial Intelligence*, pp. 1312–1317, 2011.

[11] A. Boutet, A.-M. Kermarrec, N. Mittal, and F. Taiani, "Being prepared in a sparse world: the case of KNN graph construction," in *Proceedings of the 32th IEEE International Conference on Data Engineering*, pp. 241–252, IEEE, 2016.

[12] Y. A. Malkov, A. Ponomarenko, A. Lovinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, 2013.

[13] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *Trans. PAMI*, pp. 1–1, 2018.

[14] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, pp. 509–517, Sep. 1975.

[15] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, vol. 14, (New York, NY, USA), pp. 47–57, ACM, Jun. 1984.

[16] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The x-tree : An index structure for high-dimensional data," in *Proceedings of the 22th International Conference on Very Large Data Bases*, pp. 28–39, Sep. 1996.

[17] H. Lejsek, F.-H. Asmundsson, B. B. Jonsson, and L. Amsaleg, "NV-tree: An efficient disk-based index for approximate search in very large high-dimensional collections," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 869–883, May 2009.

[18] J. Wang, N. Wang, Y. Jia, J. Li, G. Zeng, H. Zha, and X.-S. Hua, "Trinary-projection trees for approximate nearest neighbor search," *Trans. PAMI*, vol. 36, no. 2, pp. 388–403, 2014.

[19] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *Trans. PAMI*, vol. 36, pp. 2227–2240, 2014.

[20] C. Silpa-Anan and R. Hartley, "Optimised kd-trees for fast image descriptor matching," in *CVPR*, 2008.

[21] Y. Chen, T. Guan, and C. Wang, "Approximate nearest neighbor search by residual vector quantization," *Sensors*, vol. 10, pp. 11259–11273, 2010.

[22] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *Trans. PAMI*, vol. 33, pp. 117–128, Jan. 2011.

[23] A. Babenko and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *CVPR*, pp. 2055–2063, 2016.

[24] T. Zhang, C. Du, and J. Wang, "Composite quantization for approximate nearest neighbor search," in *ICML*, pp. 838–846, 2014.

[25] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, (New York, NY, USA), pp. 253–262, ACM, 2004.

[26] Q. Lv, W. Josephson, Z. Wang, and M. C. amd Kai Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search," in *Proceedings of Very Large Data bases*, Sep. 2007.

[27] W. Liu, J. Wang, R. Ji, Y. G. Jiang, and S. F. Chang, "Supervised hashing with kernels," in *CVPR*, 2012.

[28] M. Datar, P. Indyk, N. Immorlica, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Twentieth Symposium on Computational Geometry*, 2004.

[29] J. Wang, T. Zhang, J. Song, N. Sebe, and H. T. Shen, "A survey on learning to hash," *Trans. PAMI*, vol. 40, no. 4, pp. 769–790, 2018.

[30] J. Wang and S. Li, "Query-driven iterated neighborhood graph search for large scale indexing," in *Proceedings of the 20th ACM International Conference on Multimedia*, (New York, NY, USA), pp. 179–188, ACM, 2012.

[31] W. Li, Y. Zhang, Y. Sun, W. Wang, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data-experiments, analysis and improvement," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2019.

[32] W.-L. Zhao, "k-NN graph construction: a generic online approach," *Arxiv.org*, vol. abs/1804.03032, 2018. https://arxiv.org/abs/1804.03032.

[33] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," in *The Proceedings of the VLDB Endowment*, vol. 12, pp. 461–474, Jan. 2019.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2021.3101517, IEEE Transactions on Big Data

15

[34] J. V. Muoz, M. A. Gonalves, Z. Dias, and R. da S. Torres, "Hierarchical clustering-based graphs for large scale approximate nearest neighbor search," *Pattern Recognition*, vol. 96, p. 106970, 2019.

[35] B. Harwood and T. Drummond, "FANNG: Fast approximate nearest neighbour graphs," in *CVPR*, pp. 5713–5722, 2016.

[36] E. Levina and P. J. Bickel, "Maximum likelihood estimation of intrinsic dimension," *Advances in Neural Information Processing Systems*, 2005.

[37] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal on Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[38] A. F. Martin Aumller, Erik Bernhardsson, "ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," 2019.

[39] J. Sivic and A. Zisserman, "Video google: A text retrieval approach to object matching in videos," in *ICCV*, pp. 1470–1477, 2003.

[40] M. Douze, H. Jégou, H. Singh, L. Amsaleg, and C. Schmid, "Evaluation of GIST descriptors for web-scale image search," in *CIVR*, pp. 19:1–19:8, Jul. 2009.

[41] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, 2014.

[42] G. Amato, F. Falchi, C. Gennaro, and F. Rabitti, "YFCC100M hybridnet fc6 deep features for content-based image retrieval," in *Proceedings of the 2016 ACM Workshop on Multimedia COMMONS*, pp. 11–18, 2016.

[43] Q. Chen, H. Wang, M. Li, G. Ren, S. Li, J. Zhu, J. Li, C. Liu, L. Zhang, and J. Wang, *SPTAG: A library for fast approximate nearest neighbor search*, 2018.

[44] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin, "SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny," in *The Proceedings of the VLDB Endowment*, pp. 1–12, Sep. 2014.

[45] E. Bernhardsson, "Annoy: approximate nearest neighbors in C++/Python optimized for memory usage and loading/saving to disk," 2016.

[46] "OpenMP: open multiprocessing." https://www.openmp.org/.

[47] P. C. Lin and W. L. Zhao, "Graph based nearest neighbor search: Promises and failures," *arXiv.org*, 2019. https://arxiv.org/abs/1904.02077.

Peng-Cheng Lin received his Bachelor degree of Engineering from Fujian Normal University, China in 2017. He is currently a graduate student at Department of Computer Science, Xiamen University. His research interest is large-scale nearest neighbor search.

Chong-Wah Ngo received the B.Sc. and M.Sc. degrees both in computer engineering from Nanyang Technological University of Singapore, Singapore, and the Ph.D. degree in computer science from Hong Kong University of Science and Technology, Hong Kong. He is a Professor with the Department of Computer Science, City University of Hong Kong, Hong Kong. Before joining City University of Hong Kong, he was a Postdoctoral Scholar with the Beckman Institute, University of Illinois, Urbana-Champaign. His main research interests include large-scale multimedia information retrieval, video computing, multimedia mining and visualization. He was an Associate Editor for the IEEE TRANSANCTION ON MULTIMEDIA and is currently steering committee member of TRECVid, ICMR (International Conference on Multimedia Retrieval) and ACM Multimedia Asia. He is program Co-Chair of ACM Multimedia 2019, and general Co-Chairs of ICIMCS 2018 and PCM 2018. He was named ACM Distinguished Scientist in 2016 for contributions to video search and semantic understanding.

Wan-Lei Zhao received his Ph.D degree from City University of Hong Kong in 2010. He received M.Eng. and B.Eng. degrees in Department of Computer Science and Engineering from Yunnan University in 2006 and 2002 respectively. He currently works with Xiamen University as an associate professor, China. Before joining Xiamen University, he was a Postdoctoral Scholar in INRIA, France. His research interests include multimedia information retrieval and video processing.
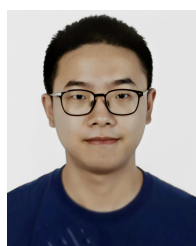
Hui Wang received his Bachelor degree of Engineering from Zhejiang Sci-Tech University, China in 2019. He is currently pursuing a Masters degree in School of Informatics at Xiamen University. His research interest is large-scale nearest neighbor search.