LiteHST: A Tree Embedding based Method for Similarity Search

YUXIANG ZENG, State Key Laboratory of Software Development Environment, Beihang University, China and The Hong Kong University of Science and Technology, China
YONGXIN TONG, State Key Laboratory of Software Development Environment, Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing, Beihang University, China
LEI CHEN, The Hong Kong University of Science and Technology, China

Similarity search is getting increasingly useful in real applications. This paper focuses on the in-memory similarity search, *i.e.*, the range query and k nearest neighbor (kNN) query, under arbitrary metric spaces, where the only known information is the distance function to measure the similarity between two objects. Although lots of research has studied this problem, the query efficiency of existing solutions is still unsatisfactory. To further improve the query efficiency, we are inspired by the tree embeddings, which map each object into a unique leaf of a well-structured tree solely based on the distances. Unlike existing embedding techniques (*e.g.*, Lipschitz embeddings and pivot mapping) for similarity search, where an extra multi-dimensional index is needed to index the embedding space (*e.g.*, L_p metrics), we directly use this tree to answer similarity search. This seems to be promising, but it is challenging to tailor tree embeddings for efficient similarity search. Specifically, we present a novel index called LiteHST, which is based on the most popular tree embedding (HST) and heavily customized for similarity search in the node structure and storage scheme. We propose a new construction algorithm with lower time complexity than existing methods and prove the optimality of LiteHST in the distance bound. Based on this new index, we also design optimization techniques that heavily reduce the number of distance computations and hence save running time. Finally, extensive experiments demonstrate that our solution outperforms the state-of-the-art in the query efficiency by a large margin.

CCS Concepts: • **Information systems** → **Proximity search**; • **Theory of computation** → *Data structures and algorithms for data management*.

Additional Key Words and Phrases: similarity search, nearest neighbor search, metric embedding

ACM Reference Format:
Yuxiang Zeng, Yongxin Tong, and Lei Chen. 2023. LiteHST: A Tree Embedding based Method for Similarity Search. *Proc. ACM Manag. Data* 1, 1, Article 35 (May 2023), 26 pages. https://doi.org/10.1145/3588715

1 INTRODUCTION

Similarity search has been widely used in numerous application scenarios, such as image recognition, analogous DNA sequence identification, and text data mining. In these applications, similarity search aims to retrieve similar objects to a given query object based on specific similarity measurements. For example, a range query finds all objects whose distances to the query object are shorter than a

Authors' addresses: Yuxiang Zeng, State Key Laboratory of Software Development Environment, Beihang University, Beijing, China and The Hong Kong University of Science and Technology, Hong Kong SAR, China, turf1013@buaa.edu.cn; Yongxin Tong, State Key Laboratory of Software Development Environment, Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing, Beihang University, Beijing, China, yxtong@buaa.edu.cn; Lei Chen, The Hong Kong University of Science and Technology, Hong Kong SAR, China, leichen@cse.ust.hk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2023/5-ART35 $15.00
https://doi.org/10.1145/3588715

given threshold and kNN query searches the k nearest neighbors to the query object, where the similarity can be measured by a distance function that satisfies the triangle inequality.

Due to the wide spectrum of applications [26, 69], many indexes, such as MVPT [16, 17], GNAT [18], BST [59], BKT [20] and SPB-tree [25], have been proposed in the past decades to achieve efficient similarity search with the exact result. We focus on the problem of in-memory similarity search in arbitrary metric spaces $S = (V, \text{dis})$ where the only known information is the distance function dis, since this setting is commonly seen in in-memory data management and has attracted much attention in recent years. For the problem that we study, unlike some existing work, we have no assumptions on (1) specific metric spaces and (2) specific representations of objects.

To perform similarity search under this setting, embedding is one of the most prevalent techniques used in such research (see surveys [23, 26, 46, 69] and books [71, 82]). In general, an embedding based method has a two-step framework. First, it maps objects into a *low-dimensional space*. Then, it indexes objects under the new embedding space (*e.g.*, by adopting a low-dimensional index). For example, the state-of-the-art embedding technique, *Lipschitz embedding* [15, 57], uses some reference objects (*a.k.a.*, pivots) to map the objects based on their distances to the reference objects. A few selected methods based on Lipschitz embeddings are SparseMap [54], MVPT [16, 17], GNAT [18], SPB-tree [25], PM-tree [72], Omni-family [58], and M-Index [65]. Other embedding techniques are mostly used in similarity search for only Euclidean metrics, such as Karhunen-Loève transform (KLT) [41] and discrete Fourier transform (DFT) [70], which are not applicable to our setting.

However, the query efficiency of these methods is still unsatisfactory. For example, there is no single method that dominates all the others in terms of query efficiency [26], since they have their pros and cons. Here, we still take the Lipschitz embedding based method as an example. Lipschitz embedding [15, 57] requires $O(\log^2 n)$ pivot sets to achieve the asymptotically optimal distance lower bound for pruning in similarity search, where n is the number of points and the *distance lower bound is crucial for query efficiency* [26, 46]. This well-known result [44, 63, 76] leads to the following dilemma in these methods. If too many pivots are used (*e.g.*, SparseMap [54] and its variants [71]), it would still be difficult to index the objects under the new embedding space (*e.g.*, L_p metrics) due to the curse of high dimensionality. For instance, when $n = 10000$, at least $\lceil \log_2 10000 \rceil^2 = 196$ pivots are used and hence a high-dimensional (196D) index is needed. By contrast, many existing methods adopt a fixed and small number of pivots, which largely deteriorates the lower bound and leads to low query efficiency.

This situation motivates us to explore a new embedding technique beyond Lipschitz embedding to design a more efficient similarity search method. Intuitively, we believe **tree embedding would be a promising answer** due to two reasons. (1) Tree embedding maps the objects into leaves of a tree-based data structure called Hierarchically Separated Tree (HST) [12]. As a result, *the high dimensional problem for Lipschitz embedding can be avoidable*, since HST itself can be used to index the objects over the embedding space and no extra index is required. (2) Prior work [35] has proved that the distance lower bound of HST is asymptotically the same as Lipschitz embedding. Unfortunately, it has never been used in similarity search, though HST has been widely used to solve combinatorial optimization problems (see the survey [56] and textbooks [44, 78]). Moreover, our experiments show that straightforwardly using HST by existing query processing methods is not always superior to the state-of-the-art method for similarity search in query efficiency. Thus, the main challenge is *how to tailor tree embeddings (HST) for efficient in-memory similarity search*?

In this paper, we propose a new tree embedding based index called LiteHST for similarity search. The **major differences** between our LiteHST and the HST [12] lie in two aspects: (1) index construction and (2) query processing. Specifically, for the *index construction*, the tree structure of LiteHST is quite different from existing HST [12, 13, 35, 80] in terms of node information and storage

scheme (see Sec. 3 for more details). Moreover, LiteHST has a significantly faster construction algorithm that takes only $O(n \log n)$ time than these existing methods (at least $O(n^2)$ time), where n is the number of objects in the input. For the *query processing*, we design many novel pruning rules based on the tree structure of LiteHST. These pruning rules are unique due to the intrinsic distinctions between tree embedding and other embeddings (*e.g.*, Lipschitz embedding), Regardless of these differences, we also prove the distance lower bound of LiteHST is no worse than that of HST (and Lipschitz embedding). Finally, motivated by the recent success of AI for DB, we also adopt the learning-based optimization technique to improve the efficiency of kNN queries.

The main contributions of this paper are summarized as follows.

- We introduce a tree embedding based index called LiteHST. Here, we design a new construction method and a compact storage scheme, and prove that LiteHST has the asymptotically same distance lower bound as the state-of-the-art embedding (*i.e.*, Lipschitz embedding) for similarity search.
- We design novel query processing methods based on LiteHST and adopt optimizations to further accelerate efficiency.
- Extensive experiments show that our solution outperforms the state-of-the-art methods in query efficiency. For example, LiteHST is up to 19.5×-34.8× faster than MVPT [16, 17], GNAT [18], BST [59], BKT [20], and SPB-tree [25] in range queries. We also show our LiteHST can also be an efficient index in the external-memory scenario.

In the rest of this paper, we first present the problem definition in Sec. 2. Then, we introduce our index (LiteHST) in Sec. 3 and query processing algorithms in Sec. 4. Finally, we conduct experiments in Sec. 5, review related work in Sec. 6, and conclude in Sec. 7.

2 PROBLEM DEFINITION AND BASELINE

In this section, we introduce the problem definition in Sec. 2.1 and briefly review the Lipschitz embedding based baselines in Sec. 2.2.

2.1 Problem Statement

Definition 1 (Metric Space [76])**.** A metric space ("metric" for short) is denoted by $S = (V, \text{dis})$, where a set of n objects is denoted by V. The distance function $\text{dis} : V \times V \rightarrow [0, +\infty)$ is used to measure the distance between any two objects in V and satisfies these properties for any object $x, y, z \in V$: (1) $\text{dis}(x, y) = 0 \Leftrightarrow x = y$, (2) $\text{dis}(x, y) = \text{dis}(y, x)$, and (3) $\text{dis}(x, y) + \text{dis}(y, z) \geq \text{dis}(x, z)$.

Such similarity metrics are commonly seen in existing studies, *e.g.*, L_p-norm and edit distance. Some related work assumes that some representations (*e.g.*, feature vectors) of objects are also given. Notice that, we do not have this assumption in the above metric.

Next, we introduce our *similarity search problem*, including the range query and k nearest neighbor (kNN) query in Def. 2-3.

Definition 2 (Range Query)**.** Given a metric $S = (V, \text{dis})$, a query object q, and a searching radius r, the range query finds all the objects in V which are within a distance of r to the object q, *i.e.*,

$$Q_{range}(V, q, r) = \{x \mid x \in V \wedge \text{dis}(x, q) \leq r\}. \quad (1)$$

Definition 3 (kNN Query)**.** Given a metric $S = (V, \text{dis})$, a query object q, and a positive integer k, the kNN query finds exactly k objects in V which are most similar to the object q, *i.e.*,

$$Q_{knn}(V, q, k) = \{x \in V' \mid V' \subseteq V \wedge |V'| = k \ \wedge \forall y \in V \setminus V', \text{dis}(q, x) \leq \text{dis}(q, y)\}. \quad (2)$$

Finally, we illustrate our similarity search problem by Example 1.

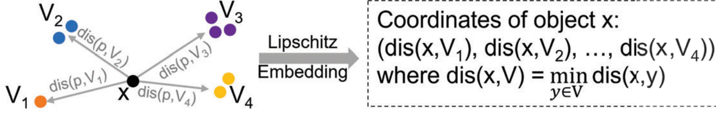


Fig. 1. The Lipschitz embedding ($R = \{V_1, \cdots, V_4\}$)

Example 1. Given a set $V = \{AGCAGCT, GCAGAGAG, AGCAGC, CGCAGA, GCAGC, AGAGAG\}$ of 6 DNA sequences, we use the edit distance to measure the similarity, *e.g.*, $\text{dis}(v_0, v_2) = 1$, where v_0 is AGCAGCT and v_2 is AGCAGC. A range query $Q_{range}(v_0, V, 1)$ retrieves two objects: v_0 and v_2, whose distances to the query object is 0 and 1 respectively, *i.e.*, closer than the searching radius $r = 1$. A kNN query $Q_{knn}(v_0, V, 2)$ finds exactly $k = 2$ objects that are most similar to the DNA of the object v_0, and the answer is v_0 and v_2.

2.2 Baseline: Lipschitz Embedding

In this subsection, we briefly introduce Lipschitz embedding [15, 57], which is commonly used in similarity search. Please refer to the surveys [5, 46] and textbooks [44, 63, 76] for more details.

Formal Definitions. We first introduce the formal definitions of the *contractive embedding*, *distortion* and *Lipschitz embedding*.

Definition 4 (Contractive Embedding and Distortion [76]). Given a metric space $S = (V, \text{dis})$, the metric $S' = (V', \text{dis}')$ is called a **contractive embedding** of S if a mapping $f : V \to V'$ exists and satisfies this condition for any object $x, y \in V$:

$$(1/\rho) \cdot \text{dis}(x, y) \leq \text{dis}'(f(x), f(y)) \leq \text{dis}(x, y), \tag{3}$$

where the **distortion** ρ measures how much shorter the distances in S' are than the corresponding distances in S, *i.e.*,

$$\rho := \max_{x,y \in V} \frac{\text{dis}(x, y)}{\text{dis}'(f(x), f(y))} \tag{4}$$

In general, an *embedding* S' aims to preserve the distances $\text{dis}(\cdot, \cdot)$ in S as much as possible. The *distortion* ρ is the standard measurement of the embedding quality [44, 63, 76]. For example, $\rho = 1$ implies that all pairwise distances in S have been perfectly preserved by S'. A *contractive embedding* is the embedding whose distances are always lower bounds of the corresponding distance in S, *i.e.*, $\text{dis}'(f(x), f(y)) \leq \text{dis}(x, y)$ in Eq. (3).

Definition 5 (Lipschitz Embedding [46]). Given a metric space $S = (V, \text{dis})$, a set $R = \{V_k\}$ of k subsets of V, a Lipschitz embedding maps S into a k-dimensional metric space $S' = (V', \text{dis}')$ such that the coordinates (*i.e.*, feature vectors) of any object $x \in V$ in S' are

$$\text{coordinate}(x) = (\text{dis}(x, V_1), \text{dis}(x, V_2), \cdots, \text{dis}(x, V_k)). \tag{5}$$

where each coordinate $\text{dis}(x, V_k)$ equals a user-defined parameter c multiplied by the distance from the object x to its closest object in V_k, *i.e.*, $\text{dis}(x, V_k) = c \cdot \min_{y \in V_k} \text{dis}(x, y)$.

As shown in Fig. 1, the Lipschitz embedding uses reference sets R (*a.k.a.*, pivot sets) to project each object $x \in V$ into a coordinate space (*e.g.*, 4D Euclidean space). A contractive embedding requires that the Euclidean distance of any two objects is no longer than their true distance and the distortion is the largest contraction ratio.

Main Idea. In similarity search, Lipschitz embedding is used to map the objects V into data points in the coordinate space S', whose distance function dis' is often an L_p distance, such as L_1, L_2 and L_∞. After that, a *filter-and-refine framework* is widely used in processing the range query and kNN query, *i.e.*, candidates are first filtered by queries on the embedding L_p metric S' and then refined by their true distances to the query object in S.

Table 1. Summary of the major notations in this paper

Notations	Descriptions
$S = (V, \text{dis})$	metric S with n objects V and distance function dis
Q_{range}, Q_{knn}	range query and kNN query in similarity search
β	a random parameter uniformly sampled in $[0.5, 1]$
T, dis_T, H	distance function dis_T on tree T with height H
$w_i = \beta 2^{H+1-i}$	edge weight between ith level and $(i+1)$th level
$\text{ball}(cp, r)$	a ball partition of V with a center cp and a radius r
lev, *par*, *child*	level, parent and children of a node v on T
dis2cp	distance from $v.cp$ to $v.par.cp$, $\text{dis}(v.cp, v.par.cp)$
$[l, r]$	object IDs from lth leaf to rth leaf on T
distort	distortion of objects in a subtree defined in Eq. (4)

We take the range query $Q_{range}(V, q, r)$ as an example, where q is the query object and r is the searching radius. When the cardinality of reference set R is 2, the objects V can be mapped into a 2D Euclidean space S' (*i.e.*, $k = 2$ and dis' is the L_2 distance). Then, we can use R-tree [71] to find all candidates $x \in V$ in S' satisfying $\text{dis}'(f(x), f(q)) \leq r$. Finally, we check each candidate whether its true distance to the query object in S, *i.e.*, $\text{dis}(x, q)$, is closer than r.

Basic Property. The *contractive property* (*i.e.*, the right-hand condition in Eq. (3)) has been proved in [46] to be necessary to guarantee no false dismissals in similarity search. The *distortion* (*i.e.*, the left-hand condition in Eq. (3)) affects the number of candidates. For example, when the distortion $\rho = 1$, the candidates are the same as the query answers. Intuitively, a lower distortion implies having fewer candidates and taking a shorter time on the refinement, which is desired in similarity search. The asymptotically lowest bound of distortion is $O(\log n)$ (with high probability) [62, 76].

Dilemma of Existing Baselines. Lipschitz embedding based methods usually have the following dilemma in the query efficiency:

(1) To achieve an optimal distortion guarantee, the Lipschitz embedding needs $\lceil \log_2 n \rceil^2$ pivot sets [46, 76]. Meanwhile, so many pivot sets imply that the dimension of the embedding space S' is still too high and causes *low efficiency in the filtering*.
(2) To retain low dimensions, many existing methods often pick fewer than 10 pivots. Hence, their distortion bound is no longer optimal, which results in much more candidates and causes *low efficiency in the refinement.*

3 OUR TREE EMBEDDING BASED INDEX

In this section, we introduces our tree embedding based index, LiteHST, including the basic structure (Sec. 3.1), construction method (Sec. 3.2), and storage scheme (Sec. 3.3). Table 1 lists the major notations used in the rest of this paper.

3.1 Definition of LiteHST

Definition 6 (Tree Embedding [12])**.** Given a metric space $S = (V, \text{dis})$, a tree embedding $S_T = (V_T, \text{dis}_T)$ is a weighted and rooted tree T, where each object $x \in V$ is mapped into a unique leaf on T (*i.e.*, $\text{leaf}(x)$) and the tree distance $\text{dis}_T(\text{leaf}(x), \text{leaf}(y))$ between objects x, y is the shortest distance from $\text{leaf}(x)$ to $\text{leaf}(y)$ on T.

Our index, LiteHST, is devised based on a popular tree embedding data structure, Hierarchically Separated Tree (HST). HST was first proposed by Bartal in his seminal work [12] and was later proved by Fakcharoenphol *et al.* [35] to have an optimal distortion guarantee ($O(\log n)$) for embedding arbitrary metrics. Although HST has been widely used in many work (see surveys [5, 56]), it

Algorithm 1: Construct our index LiteHST

```
  input  : a metric S = (V, dis) with n objects V
  output : a LiteHST T as the tree embedding of the metric S
1 Random parameter β ← uniformly sample in [0.5, 1];
2 Stack Q ← {(V, 1)} of remaining objects V at the 1st level;
3 while Q is not empty do
4     (V_j, j) ← pop remaining objects V_j at jth level from Q;
5     Ball center cp ← a random object in V_j, level i ← j + 1;
6     while V_i has more than one object (i.e., |V_i| > 1) do
7         Node u ← the objects V_i ⊆ V_{i−1} inside ball(cp, w_i);
8         Push (V_{i−1} \ V_i, i − 1) into the stack Q;
9         Process next level i ← i + 1;
```

has never been used in similarity search. Next, we introduce the *node structure* and *edge weight* of our LiteHST and clarify the difference with HST.

Node Structure. Each node v of the LiteHST is a disjoint ball partition of the objects V and stores the following information:

(1) *lev*, *par* and *child*: v's current level, parent and children.
(2) *cp*: the center of the ball partition for v.
(3) *dis2cp*: the distance from $v.cp$ to $v.par.cp$, $\text{dis}(v.cp, v.par.cp)$.
(4) $[l, r]$: the objects from the lth leaf and rth leaf, which are all inside the ball partition of v.
(5) *distort*: the distortion of these objects in the subtree rooted at v.

Edge Weight. Given a parameter β and the root of LiteHST, any edge weight between ith level and the $(i+1)$th level is defined as

$$w_1 = \beta \cdot 2^{\lceil \log_2 \max_{x \in V} \text{dis}(x, \text{root}.cp) \rceil}, \ \forall i > 2, w_i = w_{i-1}/2. \tag{6}$$

Difference with HST. One major difference between our LiteHST and classic HST [12, 35] lies in the index construction, including the node structure, construction algorithm, and storage method. For the node structure, existing work stores *lev*, *par*, *child* (only) in each node and we identify that the other information is more helpful for processing similarity search. The differences in the construction and storage will be elaborated later in Sec. 3.2 and 3.3.

3.2 Construction of LiteHST

Main Idea. LiteHST is constructed based on ball partitions of the objects V in a depth-first manner. Specifically, we first randomly pick an object *cp* from the currently remaining objects. This object *cp* is used as the center of ball partitions from the current level to the leaf level and the edge weight w_{i-1} is used as the radius of the partition at the ith level. Then, a node is created to represent the objects inside the ball while the other objects outside the ball are waiting for further partitions.

Algorithm Details. Algo. 1 depicts the construction procedure. Specifically, line 1 samples a random parameter β from a uniform distribution. Line 2 initializes a stack Q of the remaining objects at each level. In lines 3-9, we construct a tree embedding T of the input metric (V, dis). Specifically, we first pop the remaining objects V_j at the jth level from Q in line 4. Then, we randomly select an object $cp \in V_j$ as the center of the ball partition in line 5. Next, at each level i, we use V_i to denote the objects in V_{i-1} that are inside the ball centered at cp with a radius w_i (denoted by $\text{ball}(cp, w_i)$) in line 7. After that, we create a node u to represent these objects V_i and push the remaining objects $V_{i-1} \setminus V_i$ at the $(i-1)$th level into Q in line 8. The ball partitions in

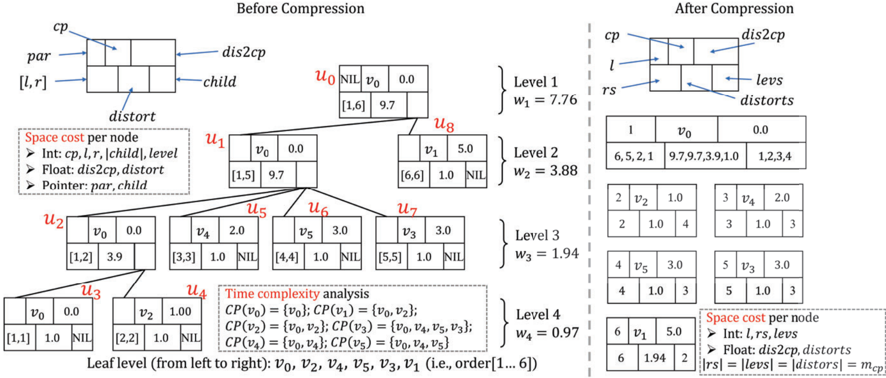


Fig. 2. An instance of our index LiteHST (left: pointer-based; right: array-based)

lines 6-9 will stop when $|V_i| = 1$, since a leaf in LiteHST represents a unique object (*i.e.*, the center *cp*). Finally, we maintain nodes' information (defined in Sec. 3.1).

Example 2. Back to Example 1. Fig. 2 illustrates a LiteHST when the random parameter $\beta = 0.97$. We assume the object v_0 is firstly sampled as the center *cp* in line 5 of Algo. 1. By Eq. (6), we have $w_1 = \beta \times 2^{\lceil \log_2 \text{dis}(v_0, v_1) \rceil} = 7.76$ and $w_2 = w_1/2$. Then, when $i = 2$ during the iterations in lines 6-9, the objects V_2 is v_0 and v_2-v_5. Here, v_1 is excluded, since $\text{dis}(cp, v_1) = 5 > w_2$ indicates v_1 is not in the ball partition $\text{ball}(cp, w_2)$. Thus, we create the internal node u_1 to represent V_2 and build the subtree rooted at u_1. In the internal node u_1, the attribute of $[l, r]$ is $[1, 5]$, which means the 1st leaf to the 5th leaf (*i.e.*, objects V_2) are covered by the ball partition of this node. Besides, $u_1.distort$ can be calculated by Eq. (4), *i.e.*, $u_1.distort = \max_{x,y \in V_2} \frac{\sum_{e \in path(x,y)} W(e)}{\text{dis}(x,y)} = \frac{2 \times 3.88 + 1.94}{\text{dis}(v_2, v_4)} = 9.7$. Similarly, we can derive the other attributes in Fig. 2.

Complexity Analysis. Let the function dis(,) take $O(d)$ time. For each remaining object set V_j in line 4, lines 6-8 involve $O(|V_j| \cdot d)$ distance computations in total, since we only need to compute the distance from the center *cp* to each object in V_j for once. For each object $x_i \in V$, we use $CP(x_i)$ to denote the list of centers whose distances to x_i have been computed. For example, as shown in Fig. 2, $CP(v_1) = \{v_0, v_1\}$, since $\text{dis}(v_0, v_1)$ and $\text{dis}(v_1, v_1)$ have been computed when the leaf node u_8 (*i.e.*, $\text{leaf}(v_1)$) is created. Thus, the time complexity of lines 1-9 is bounded by $O(d \sum_{i=1}^{n} |CP(x_i)|)$. To obtain a tight bound, we borrow the following fact from [14].

Fact 1 (Lemma 2.1 in [14]). *Given a random permutation π of n integers, let a non-increasing y_i be $\min_{j=1,\cdots,i}\{\pi[j]\}$. We have (1) the number of times in y_i to change into a smaller value is $O(\log n)$ in expectation and (2) it is $O(\log n)$ with high probability.*

By using this fact, we know the expected length of $CP(x_i)$ is $O(\log n)$ with high probability (*i.e.*, $CP(x_i)$ corresponds to y_i), since $CP(x_i)$ contains at most n centers that are uniformly sampled in Algo. 1, and the last center in $CP(x_i)$ is x_i itself (*i.e.*, $\text{leaf}(x_i)$). Thus, the (expected) time complexity of lines 1-9 is $O(dn \log n)$. Besides, the distortion of n objects defined in Eq. (4) takes $O(n^2)$ time. To reduce this time cost, we only compute the distortions of those subtrees with $O(\log n)$ objects. Overall, the (expected) time complexity of Algo. 1 is still $O(dn \log n)$.

Approximation Analysis. We prove Algo. 1 has a tight theoretical guarantee (defined in Def. 7) on the distortion, *i.e.*, $O(\log n)$.

Definition 7 (Probabilistic Approximation [12]). A tree embedding S_T is said to be ρ-probabilistically approximates the metric space S if there exists a probability distribution over S_T such that for any two objects $x, y \in V$ we have $\text{E}[\text{dis}_T(\text{leaf}(x), \text{leaf}(y))] \le \rho \cdot \text{dis}(x, y)$

Lemma 1. *Algo. 1 has a tight distortion guarantee $O(\log n)$.*

Proof. We prove this lemma by *mathematical induction* based on the approximation result from [35], since Fakcharoenphol *et al.* [35] have proved that the HST constructed by their algorithm FRT has a *tight* distortion guarantee of $O(\log n)$. We assume both methods are given the same parameter β for edge weights, since the distributions of β are the same. Thus, we have the distance functions for both methods are $\mathrm{dis}_T(u,v) = \sum_{e \in path(u,v)} w(e)$, where u, v are two leaves of the (sub)tree rooted at $root$, $path(u,v)$ is the tree path from u to v, and $w(e)$ is the weight of the edge e.

When there is only one object, either Algo. 1 or FRT constructs the same tree (*i.e.*, a single root).

When there is more than one object, we assume $O(\log n)$ holds for $n = m$ and prove it also holds for $n = m + 1$. Specifically, we use cp_1 to denote the first center sampled by Algo. 1 in line 5. Then, the number of remaining objects $V_{i-1} \setminus V_i$ in line 8 is no more than $n - 1 = m$, since a leaf will be eventually created to represent the object cp_1. It implies that the distortion guarantees of subtrees for the remaining objects are all bounded by $O(\log m) = O(\log n)$. It remains to be proved that $\mathrm{E}[\mathrm{dis}_T(\mathrm{leaf}(x), \mathrm{leaf}(cp_1))] \leq \rho \cdot \mathrm{dis}(x, cp_1)$, where the distortion $\rho = 96 \ln n = O(\log n)$.

Note that the centers of FRT [35] are sequentially picked from a global and random permutation π of all objects V. It means its first center (say cp'_1) is also uniformly sampled from V. Thus, our center cp_1 has the same distribution as the first center cp'_1 by FRT. Since FRT has proved that $\mathrm{E}[\mathrm{dis}_T(\mathrm{leaf}(x), \mathrm{leaf}(cp'_1))] \leq 96 \ln n \cdot \mathrm{dis}(x, cp'_1)$ (see Sec. 2.3 in [35]), we complete our proof. □

Difference with HST. Compared with existing construction algorithms [12–14, 35, 42, 60, 80] that take at least $O(dn^2)$ time, our algorithm has a significantly lower time complexity ($O(dn \log n)$).

3.3 Storage of LiteHST

Motivation. LiteHST can be resident in memory with a pointer-based storage scheme. However, we observe this pointer-based storage scheme has information redundancy, which is about 36%-50% of the total space cost. As shown in Fig. 2, the left-most descendants (*e.g.*, u) of any node must have the same center and hence $u.cp = u.par.cp$ and $u.dis2cp = 0.0$ always holds. To reduce this information redundancy and save the space cost, we propose an array-based storage scheme for LiteHST as follows.

Main Idea. In Algo. 1, each object is used as the center for once. Thus, our idea is to store the index information from the view of each center. For example, we use an array $row[\cdot]$ to store the index information and $row[cp]$ stores the following information lead by ball partitions centered at cp (in lines 5-9 of Algo. 1):

(1) *dis2cp*: distance from cp to the center of V_j's parent.
(2) l: the position of $\mathrm{leaf}(cp)$ at the leaf level.
(3) *rs*: a list of the right boundary (r) for nodes with center cp.
(4) *levs*: a list of levels (lev) for nodes with center cp.
(5) *distorts*: a list of distortions (*distort*) for nodes with center cp.

Algorithm Details. Based on these definitions, a pointer-based LiteHST can be transformed into an array-based LiteHST. The *main issue* is how to enumerate the child nodes in this array-based HST, since our storage scheme does not involve pointers referring to children. Algo. 2 depicts the detailed procedure. Suppose we want to enumerate all children of the node partitioned by the center cp at the kth level. In lines 1-3, we observe that the left-most child (u_1) must be partitioned by $cp' = cp$ at the level $row[cp'].levs[i]$. Based on the right boundary $row[cp'].rs[i]$, we can infer the next sibling u_2 of u_1 must be partitioned by the object which is right after $row[cp'].rs[i]$ in the leaf level. We use $\mathrm{order}[\cdot]$ to denote the sequence of corresponding objects in the leaf level. Thus, line 3 iterates each child node, and line 4 moves to the next child.

Algorithm 2: Enumerating the children of the node partitioned by the center cp at the kth level

1. $l \leftarrow row[cp].l$, $cp' \leftarrow cp$, $i \leftarrow k+1$;
2. **while** $l \leq$ *right boundary* $row[cp].rs[k]$ *of parent node* **do**
3.   next child is partitioned by cp' at level $row[cp'].levs[i]$;
4. $l \leftarrow row[cp'].rs[i]+1$, $cp' \leftarrow$ order$[l]$, $i \leftarrow 0$;

Example 3. An array-based LiteHST is created on the right side of Fig. 2. To enumerate the children of the root u_0 whose center is v_0, we set $l = 1$, $cp' = v_0$, $i = 2$ in line 1. Then, in line 3, the first child can be derived from $row[cp']$ at the 2nd level, which corresponds to u_1 in Fig. 2. After that, we have $l = row[v_0].rs[2] + 1 = 6$, $i = 0$ and cp' becomes the 6th object in the leaf level (*i.e.*, v_1). Finally, we access the sibling (u_8) of the node u_1 via $row[v_1]$.

Difference with HST. Compared with the pointer-based scheme used in traditional HST [12, 35], our array-based storage scheme can save 36%-50% spaces based on the following *space-saving analysis*. First, we assume an integer type takes 4 bytes, a floating type takes 8 bytes and a pointer type takes 4 types (*e.g.*, in C/C++). Let U and E denote the nodes and edges on LiteHST. In a *pointer-based* storage scheme, each node stores 2 floating types (*dis2cp* and *distort*), $1 + |child|$ pointer types (*par* and *child*), and 5 integer types ($|child|$, *lev*, *cp* and $[l, r]$). Thus, we can derive the space cost as

$$2 \times 8 \times |U| + 4 \times (|U| + |E|) + 5 \times 4 \times |U| = 40|U| + 4|E| \tag{7}$$

As for the *array-based* storage scheme, let m_{cp} be the number of nodes with the center cp, *i.e.*, $|rs| = |levs| = |distorts| = m_{cp}$. As each node has only one center, we know $\sum_{cp \in V} m_{cp} = |U|$. As shown in Fig. 2 (bottom right-hand corner), $row[cp]$ stores $1+m_{cp}$ floating types (*dis2cp* and *distorts*) and $1 + 2m_{cp}$ integer types (*l*, *rs* and *levs*). Thus, the space cost of this new storage scheme is

$$\sum\nolimits_{cp \in V} \left(8(1 + m_{cp}) + 4(1 + 2m_{cp})\right) = 12|V| + 16|U| \tag{8}$$

Finally, since each internal node has at least two children and LiteHST has $|V| = n$ leaves, we have $|U| = |E| + 1$ and $|U| \in [n, 2n]$, we can easily derive the space saving is 36%-50% as follows.

$$1 - \frac{12|V| + 16|U|}{40|U| + 4|E|} \approx 1 - \frac{12n + 16|U|}{44|U|} = \frac{7}{11} - \frac{3}{11} \cdot \frac{n}{|U|} \tag{9}$$

Remark. Although our major concern is an in-memory index for similarity search, LiteHST can still be extended to the external-memory scenario. Specifically, we can still use the above construction algorithm and storage scheme. The main change is that a leaf node in the external-memory scenario will be created when the number of contained objects could fit in one disk page. Similar to other external-memory indexes [25, 58], we also utilize the random access file (RAF) to store these objects in disks. Each RAF entry records the actual object and its distance to the center of the leaf node. The RAF stores these records in an ascending order of (1) their leaf nodes' appearance orders (from left to right) and (2) the distances to the leaf node' center (for objects in the same leaf).

4 QUERY PROCESSING METHODS

In this section, we first introduce the motivation and preliminary in Sec. 4.1. Then, we present our solutions to range queries and kNN queries in Sec. 4.2 and Sec. 4.3, respectively. Finally, we discuss our learning-based enhancement for query efficiency in Sec. 4.4. The following query processing methods, which are designed for in-memory LiteHST, are also applicable to the external-memory

LiteHST. The main difference is that the external-memory LiteHST needs to scan more than one object that is stored in a leaf node. All proofs in Sec. 4.2 and 4.3 are deferred to Sec. 4.5.

4.1 Motivation and Preliminary

Motivation. As explained in Sec. 3, LiteHST can be viewed as not only a contractive embedding of the input metric but also a ball partition of this metric. Thus, two existing solutions for similarity search can be directly used for LiteHST: *M-tree based solution* [27] and *contractive-embedding based solution* [47] (see Sec. 5.1.1 for the implementations). However, neither of them outperforms the state-of-the-art in our experiments, since they all fail to fully utilize the properties of LiteHST. By contrast, this section presents a new solution that is faster than the existing ones.

Preliminary. Following primitives, which only take $O(1)$ time, are used in our solution.

(1) $\mathsf{radAt}(i)$: the radius of the ball partition at the ith level, *i.e.*, $\mathsf{radAt}(i) = w_{i-1}$, where w_{i-1} is defined in Eq. (6).
(2) $\mathsf{disTAt}(i)$: the sum of edge weights between two leaves whose lowest common ancestor is at level i, *i.e.*, $\mathsf{disTAt}(i) = 2\sum_{j=i}^{H} w_i$.
(3) $\mathsf{ring}(o, r, R)$: a ring centered at the object o with an inner radius r and an exterior radius R, where $r \leq R$.
(4) $\mathsf{coverby}(o, r_1, R_1, r_2, R_2)$: it checks whether $\mathsf{ring}(o, r_1, R_1)$ is covered by $\mathsf{ring}(o, r_2, R_2)$, where both rings are centered at o.
(5) $\mathsf{overlap}(o, r_1, R_1, r_2, R_2)$: it checks whether $\mathsf{ring}(o, r_1, R_1)$ overlaps with $\mathsf{ring}(o, r_2, R_2)$, where both rings are centered at o.

4.2 Range Query Processing

Main Idea. To answer range queries, we use a breadth-first search in the LiteHST and classify any searched node v into three kinds:

(1) The v's covered objects (*i.e.*, from $v.l$th leaf to $v.r$th leaf) can be directly added to the query answer, when they are guaranteed to be *inside* the query range (by Lemma 2 (1) and (3)).
(2) The v's covered objects (*i.e.*, from $v.l$th leaf to $v.r$th leaf) can be ignored, when they are guaranteed to be *outside* the query range (by Lemma 2 (2), (4)-(6) and Lemma 3).
(3) Otherwise, we need to search the children of this node v.

Pruning Strategy. Let v denote the current node during the breadth-first search and $\mathsf{dis2q} = \mathsf{dis}(v.cp, q)$ be the distance from its center $v.cp$ to the query object q. The basic idea of our pruning strategies is as follows. We first derive the lower bound (LB) and upper bound (UB) of the distances from $v.cp$ to any object inside the query range (no matter whether it is in V or not). Then, for this node v, its child or its right sibling, we derive the lower bound (lb) and upper bound (ub) of the distances from $v.cp$ to any object covered by u. Finally, we use these bounds and primitive operations to determine the right kind for each node. All these bounds are derived based on (1) properties of metric space (in Def. 1), (2) structures of our index LiteHST (in Sec. 3.1), and (3) the construction method (in Sec. 3.2).

When an internal node v is searched, Lemma 2 is used to prune its child nodes. When some child nodes $(u_1, u_2, \cdots, u_{i-1})$ of v are also searched, Lemma 3 is used to prune their sibling nodes.

Lemma 2. *Given a range query $Q_{range}(V, q, r)$, a node v, v's non left-most child u, and $dis2q = dis(v.cp, q)$, our pruning strategy is*
(1) v's covered objects are in the query range if $dis2q + radAt(v.lev) \leq r$;
(2) v can be pruned if $overlap(v.cp, LB, UB, 0, radAt(v.lev))$ is false, where $LB = \max\{0, dis2q - r\}$, $UB = \min\{radAt(v.lev), dis2q + r\}$;
(3) u's covered objects are in the query range if $dis2q + \min\{radAt(v.lev), u.dis2cp + radAt(u.lev)\} \leq r$;

Algorithm 3: Answer exact range query

```
   input  : a range query Q_range(V, q, r) and LiteHST T
   output : the exact answer (denoted by ans)
1  rt ← T's root, queue Q ← {(rt, dis(rt.cp, q))};
2  while Q is not empty do
3      (rt, dis2q) ← pop from the head of Q;
4      foreach node v ∈ {rt} ∪ {rt's left-most descendants} do
5          if Lemma 2 (1) is satisfied then
6              add all v's objects into ans and break;
7          if Lemma 2 (2) is satisfied then break;
8          if Lemma 2 (6) is satisfied then continue;
9          dis2q* ← dis2q, u* ← v's left-most child;
10         foreach node u ∈ non left-most children of v do
11             if Lemma 2 (3) is satisfied then
12                 add all u's objects into ans and continue;
13             if Lemma 2 (4) and Lemma 3 (1) are violated then push (u, dis(u.cp, q)) to the tail
                of Q and maintain u*, dis2q* based on Lemma 3;
14             if Lemma 3 (2) is satisfied then break;
15         if Lemma 2 (5) is satisfied then break;
```

(4) u can be safely pruned if $overlap(v.cp, LB, UB, lb, ub)$ is false, where $lb = \max\{radAt(v.lev+1), disTAt(v.lev)/v.distort, u.dis2cp - radAt(u.lev)\}$, $ub = \min\{radAt(v.lev), u.dis2cp + radAt(u.lev)\}$;
(5) v's left-most child node can be pruned if $radAt(v.lev+1) < LB$;
(6) v's other child nodes can be pruned if $radAt(v.lev+1) > UB$.

Lemma 3. *Given a range query $Q_{range}(V, q, r)$, an internal node v, its child nodes $u_1, u_2, \cdots, u_k$ (from left to right), we have searched the first $i-1$ child nodes and define $u^* = \arg\min_{j<i} dis(u_j.cp, q)$ and $dis2q^* = dis(u^*.cp, q)$. Our pruning strategy is:*
(1) u_i can be pruned if $overlap(u^*.cp, LB, UB, lb, ub)$ is false, where the bounds $LB = \max\{0, dis2q^* - r\}$, $UB = \min\{2radAt(v.lev), dis2q^* + r\}$, $lb = \max\{radAt(v.lev+1), disTAt(v.lev)/v.distort, |u^*.dis2cp - u_i.dis2cp| - radAt(u_i.lev)\}$, and $ub = u^*.dis2cp + \min\{radAt(v.lev), u_i.dis2cp + radAt(u_i.lev)\}$;
(2) u_i and its right sibling nodes can be pruned if $\max\{radAt(v.lev+1), disTAt(v.lev)/v.distort\} - r > dis2q^*$.

Algorithm Details. In Algo. 3, we use a queue Q to maintain the next internal node to be searched. Lines 4-15 search the subtree rooted at the rt whose center to the query object q is dis2q. Specifically, we use v to denote rt itself or any rt's left-most descendants. Since all the iterated nodes v have the same center object, we have dis2q = dis$(v.cp, q)$. If the condition of line 5 is satisfied, then all the objects contained in v must be inside the query range. Lines 7-8 testify whether the objects contained in v and its left siblings are outside the query range. In lines 9-15, we enumerate each non left-most child node of u and put it into Q when its covered objects may be in the query range. Line 11 puts all u's covered objects into the answer ans, if the corresponding circular range of u is covered by the query range. Lines 13-15 are based on Lemmas 2 and 3. In line 13, dis2q* and u^*, which are used to prune u's right siblings (*i.e.*, break the inner loop of line 10), are maintained by Lemma 3. Line 15 is used to prune v's right siblings.

Example 4. Back to Example 1. We want to use the LiteHST in Fig. 2 to answer the range query $Q_{range}(v_0 = AGCAGCT, V, 1)$. The searching procedure of Algo. 3 starts from the root u_0. In lines 4-15, we try to prune some unqualified nodes which have no objects within the query range. For example, when dis2q = dis$(u_0.cp, v_0) = 0$, we can safely prune the node u_8 due to Lemma 2 (6). Specifically, we first compute $UB = \min\{\text{radAt}(u_0.lev), \text{dis2q} + r\} = \min\{w_1, 0+1\} = 1$. Thus, we have radAt$(u_0.lev + 1) = w_2 > UB$ and Lemma 2 ensures that no object covered by u_9 belongs to the query answer.

4.3 *k*NN Query Processing

Main Idea. For *k*NN queries, we use a *best-first-search* algorithm with an *adaptively decreased searching radius* (denoted by r_k). During the search, we still use the pruning strategy for a range query $Q_{range}(V, q, r_k)$ to get the candidate objects. When the searching radius r_k is decreased to the *k*th nearest distance to the query object q, we will eventually find the k nearest neighbors.

Specifically, a heap Q is used to organize the searching orders of the nodes (say u) based on the *lower bound dmin* of distances from the query object q to u's covered objects, *i.e.*, $dmin = \{0, \text{dis}(u.cp, q) - \text{radAt}(u.lev)\}$. We can safely terminate the search procedure when the top element in the heap Q is larger than the radius r_k.

Besides, the radius r_k is maintained with the longest distance in a distance set NN_{dis}, which has at most k distances of different objects in V to the query object q. Specifically, for a node u, *dmax* is used to denote the *upper bound* of distances from the query object q to u's covered objects, *i.e.*, $dmax = \text{dis}(u.cp, q) + \text{radAt}(u.lev)$ (u is an internal node) and $dmax = \text{dis}(u.cp, q)$ (u is a leaf node). Then, we can put at most $\min\{k, u.rs - u.l + 1\}$ objects into NN_{dis} with the distance upper bound *dmax* and maintain r_k accordingly. Once an internal node has been popped from Q, we also remove the corresponding upper bounds in NN_{dis}.

Pruning Strategy. Except for Lemma 2 and Lemma 3, we propose another pruning strategy in Lemma 4 for processing *k*NN queries.

Lemma 4. *Given a kNN query $Q_{knn}(V, q, k)$, an internal node p, p's child nodes $v_1, v_2, \cdots, v_k$ (from left to right), v_1's left-most child node u_1, and dis2q = dis$(p.cp, q)$, our pruning strategy is as follows. The nodes v_2-v_k can be pruned if dis2q $\leq$ radAt$(u_1.lev + 1)$ and the number of objects in u_1 is no smaller than k (i.e., $u_1.r - u_1.l + 1 \geq k$).*

Algorithm Details. In line 1 of Algo. 4, we compute the distance (dis2q) from q to the root's center object. We also calculate the lower/upper bound (*dmin*/*dmax*) of distances from q to the roots' covered objects. We also use r_k to denote the searching radius. After that, we push the tuple $(rt, \text{dis2q}, dmin, dmax)$ into a min-heap Q, where the tuples in Q are maintained based on *dmin*. We add the tuple $(rt, dmax, k)$ into a distance set NN_{dis}, where k denotes at least k objects in rt that have distances to q within *dmax* and the tuples in NN_{dis} are maintained by *dmax*. Lines 2-14 depict the searching procedure. Line 3 pops the top tuple $(v, \text{dis2q}, dmin, dmax)$ from the top of Q. Iterations in lines 4-14 search each child node u of v. Here, the pruning strategies (*i.e.*, Lemma 2 and Lemma 3) for range query processing can be also used in these iterations. If u is not pruned, we will calculate dis2q, *dmin*, *dmax* with respect to u in lines 6-10. If $dmin \leq r_k$, we will update Q, NN_{dis}, and r_k in lines 12-14. In line 15, the exact answer *ans* is obtained from the objects in NN_{dis}.

4.4 Learning based Optimization

Main Idea. Intuitively, a smaller r_k in Algo. 4 implies faster processing for a *k*NN query. Thus, we design a learning-based optimization to derive a small enough initialization (denoted by $\widetilde{r_k}$) of the searching radius r_k. By this way, we can save the time cost, since the above pruning strategies are more likely to work when the radius is short. Specifically, our optimization works as follows.

(1) We use the regression model to predict the position *qid* of the query object q at the leaf level.

Algorithm 4: Answer exact kNN query

```
  input  : a kNN query Q_knn(V, q, k) and LiteHST T
  output : the exact answer (denoted by ans)
1  rt ← T's root, dis2q ← dis(rt.cp, q), dmin ← max{0, dis2q − radAt(rt.lev)},
     dmax ← dis2q + radAt(rt.lev), r_k ← dmax, Q ← {(rt, dis2q, dmin, dmax)},
     NN_dis ← {(rt, dmax, k)};
2  while Q is not empty and top element's dmin ≤ r_k in Q do
3      (v, dis2q, dmin, dmax) ← pop from the top of Q;
4      foreach child node u of the node v do
5          Use Lemma 2 and Lemma 3 to prune u for a circular range query
             Q_range(u's objects, q, r_k);
6          dis2q ← dis(u.cp, q);
7          if u is an internal node then
8              dmax ← dis2q + radAt(u.lev);
9              dmin ← max{0, dis2q − radAt(u.lev)};
10         else dmax ← dis2q, dmin ← dis2q ;
11         if dmin ≤ r_k then
12             Push (u, dis2q, dmin, dmax) into min-heap Q;
13             Add (u, dmax, u.r − u.l + 1) into NN_dis;
14             r_k ← the kth longest distance in NN_dis;
15 ans ← the objects contained in the nodes in NN_dis;
```

(2) We use our estimation algorithm (*i.e.*, Algo. 5) to derive a suitable initial value $\widetilde{r_k}$ based on *qid* and k for the kNN query.
(3) Finally, we set r_k with $\widetilde{r_k}$ and run Algo. 4.

Although a learning model may involve errors, the following lemma guarantees the correctness of our solution to kNN queries.

Lemma 5. *The estimation $\widetilde{r_k}$ in line 5 of Algo. 5 is an upper bound of the kth nearest distance to the query object q.*

Proof. Let V' be the objects covered by the node v in line 5 of Algo. 5. Due to the construction of Lite-HST, we have $\forall x \in V', \mathrm{dis}(v.cp, x) \leq \mathrm{radAt}(v.lev)$. By the triangle inequality, we know $\mathrm{dis}(q, x) \leq \mathrm{dis}(q, v.cp) + \mathrm{dis}(v.cp, x) \leq \mathrm{dis}(v.cp, q) + \mathrm{radAt}(v.lev)$. Since V' has at least k objects (line 4), we can derive that the estimation $\widetilde{r_k}$ is no shorter than the kth nearest distance. □

Algorithm Details. For the *regression*, the *main challenge* is how to derive the feature vectors for each input object in V, since the metric studied in our paper may not be a coordinate space (*e.g.*, Euclidean space). To tackle this challenge, we borrow the idea of Lipschitz embeddings defined in Sec. 2.2. Specifically, we select m objects in V as m reference sets in Lipschitz embeddings, where each reference set has only one object. After that, we create a m-dimensional coordinate as feature vectors of each object, and the label is its position in the leaf level. To pick these m objects, we first compute the total number of objects have been covered by each object in V in all levels of a LiteHST and then choose the top-m objects that have the largest total number. Finally, existing regression model can be used to predict the mapping position in the leaf level for each new object, *e.g.*, Gradient Boosting Regression Tree (GBRT) [45] with $m = 15$ is used in our experiments.

Algorithm 5: Estimate the kth nearest distance

input : a kNN query $Q_{knn}(V, q, k)$ and LiteHST T
output: an estimation $\widetilde{r_k}$ of the kth nearest distance to q

1 $qid \leftarrow$ predict the position of q at the leaf level;
2 $v \leftarrow$ the qidth leaf on LiteHST T;
3 **while** v *is not empty* **do**
4   **if** v *covers more than* k *objects* **then**
5         $\widetilde{r_k} \leftarrow \mathsf{dis}(v.cp, q) + \mathsf{radAt}(v.lev)$, **break**;
6     $v \leftarrow$ the parent of v;

For the *estimation*, Algo. 5 illustrates the details. In lines 1-2, we first predict the position qid of the query object q at the leaf level and use v to denote the qidth leaf. Lines 3-6 traverse the ancestors of this leaf node. If this ancestor covers more than k objects, we estimate the kth nearest distance in line 5 and break the iteration. The estimation $\widetilde{r_k}$ is an upper bound of kth nearest distance.

Example 5. Back to Example 1. We try to answer a kNN query $Q_{knn}(AGCAGCT, V, 2)$. If we can predict the accurate position of the query object v_0 = AGCAGCT in the leaf level (*i.e.*, $qid = 1$ in Fig. 2), we can easily infer that $v = u_2$ covers no fewer than 2 objects in line 4. Thus, in line 5, we can estimate the initial value of r_k is $\widetilde{r_k} = \mathsf{dis}(u_2.cp, q) + \mathsf{radAt}(u_2.lev) = 0.0 + 1.94 = 1.94$.

Remark. The parameter m can be tuned based on the intrinsic dimensionality of the datasets. For example, we can tune this parameter by slightly changing the values of m around the intrinsic dimensionality and observing the query efficiency of a random query workload. This is because existing research [25, 58] has suggested to use the intrinsic dimensionality to be the number of reference points in Lipschitz embeddings. The intrinsic dimensionality can be computed by two ways: *global methods* and *local methods*. Their main difference is that a global method provides an estimation for all objects while a local method computes the estimation for partial objects. For example, a popular global method [23] uses μ^2/ω^2 as the intrinsic dimensionality, where μ and ω are the mean and variance of pairwise distances. By contrast, local methods [6, 7] estimate the local intrinsic dimensionality (LID) [48–51] of partial objects. According to [21], the (global) intrinsic dimensionality can be estimated by averaging the LID values over large enough samples. The local methods are relatively efficient and can be applied here due to the result in Lemma 5.

4.5 Deferred Proofs of Lemmas 2-4

The proof of **Lemma 2** is as follows.

Proof. Let x be an object contained in v and y be an object contained in its child node u. Based on the node structure and construction algorithm, we know $\mathsf{dis}(v.cp, x) \leq \mathsf{radAt}(v.lev)$, $\mathsf{dis}(v.cp, y) \leq \mathsf{radAt}(v.lev)$, $\mathsf{dis}(u.cp, y) \leq \mathsf{radAt}(u.lev)$, $u.dis2cp = \mathsf{dis}(v.cp, u.cp)$.

Case (1). We derive the *upper bound* of $\mathsf{dis}(q, x)$ as

$$\mathsf{dis}(q, x) \leq \mathsf{dis}(v.cp, q) + \mathsf{dis}(v.cp.x) \leq \mathsf{dis2q} + \mathsf{radAt}(v.lev).$$

The prerequisite of this case is $\mathsf{dis2q} + \mathsf{radAt}(v.lev) \leq r$, which means $\mathsf{dis}(q, x) \leq r$ and x is inside the query range.

Case (2). Let LB and UB denote the lower bound and upper bound of distances between $v.cp$ and any object z that is inside the query range and contained in v. As z is in the query range, we know $\mathsf{dis}(q, z) \leq r$. By the triangle inequality, we have $\mathsf{dis}(v.cp, z) \geq \mathsf{dis}(v.cp, q) - \mathsf{dis}(q, z) \geq \mathsf{dis2q} - r$ and $\mathsf{dis}(v.cp, z) \leq \mathsf{dis}(v.cp, q) + \mathsf{dis}(q, z) \leq \mathsf{dis2q} + r$. Since z is contained in v, we know

$\text{dis}(v.cp, z) \leq \text{radAt}(v.lev)$. Now, we have proved the correctness of LB and UB. We next derive the *lower bound* of $\text{dis}(q, x)$ as

$$\text{dis}(q, x) \geq \text{dis}(v.cp, q) - \text{dis}(v.cp.x) \geq \text{dis2q} - \text{radAt}(v.lev).$$

The prerequisite of this case is $\text{radAt}(v.lev) < LB < \text{dis2q} - r$, which means $\text{dis}(q, x) > r$ and x is not in the query range.

Case (3). We derive the *upper bound* of the distance $\text{dis}(q, y)$ as:

$$\text{dis}(q, y) \leq \text{dis}(q, v.cp) + \text{dis}(v.cp, y) \leq \text{dis2q} + \text{radAt}(v.lev)$$

$$\begin{aligned}\text{dis}(q, y) &\leq \text{dis}(q, v.cp) + \text{dis}(v.cp, u.cp) + \text{dis}(u.cp, y) \\ &\leq \text{dis2q} + u.dis2cp + \text{radAt}(u.lev)\end{aligned}$$

The prerequisite of this case is $\text{dis2q} + \text{radAt}(v.lev) \leq r$ and $\text{dis2q} + u.dis2cp + \text{radAt}(u.lev) \leq r$, which means $\text{dis}(q, y) \leq r$ and y is inside the query range.

Case (4). lb and ub denote the lower bound and upper bound of distance from $v.cp$ to y. We first show the correctness of lb. As u contains y and u is not v's left-most child, we have $\text{dis}(v.cp, y) > \text{radAt}(v.lev + 1)$ (otherwise, the left-most child is u). Since v is the lowest common ancestor of the leaves of $v.cp$ and y, we have $Dis_T(y, v.cp) = \text{disTAt}(v.lev)$ by Def. 6. Based on Def. 4, we know $\text{dis}(y, v.cp) \geq Dis_T(y, v.cp)/v.distort \geq \text{disTAt}(v.lev)/v.distort$. By the triangle inequality, we have $\text{dis}(y, v.cp) \geq \text{dis}(v.cp, u.cp) - \text{dis}(y, u.cp) \geq u.dis2cp - \text{radAt}(u.lev)$.

We next prove the correctness of ub. Similar to that of lb, we have $\text{dis}(y, v.cp) \leq \text{dis}(v.cp, u.cp) + \text{dis}(y, u.cp) \geq u.dis2cp + \text{radAt}(u.lev)$. Since v contains y, $\text{dis}(y, v.cp) \leq \text{radAt}(v.lev)$.

We finally prove the statement of this case. The prerequisite of this case is that $\text{ring}(v.cp, LB, UB)$ does not overlap with $\text{ring}(v.cp, lb, ub)$. Due to the definitions of these lower/upper bounds, the object y cannot be inside the query range,

Case (5). This case is a corollary of the second case. Let v' be the left-most child node of v. Based on the construction algorithm, we know $\text{radAt}(v'.lev) \leq \text{radAt}(v.lev + 1)$ and $v'.cp = v.cp \implies \text{dis2q}' = \text{dis}(v'.cp, q) = \text{dis2q}$. The prerequisite of this case is $\text{radAt}(v.lev + 1) < LB$, where $LB = \max\{0, \text{dis2q} - r\}$. Thus, we have $\text{radAt}(v'.lev) < \max\{0, \text{dis2q}' - r\}$. We complete the proof by substituting v', $\text{dis2q}'$ for v, dis2q in the second case.

Case (6). This case is a corollary of the fourth case. Based on the prerequisite, we can infer that $u2vLB \geq \text{radAt}(v.lev + 1) > UB$. Thus, $\text{overlap}(v.cp, LB, UB, u2vLB, u2vUB)$ is always false. □

The proof of **Lemma 3** is as follows.

Proof. Let x be an object contained in u_i. Based on the construction algorithm of LiteHST, we know $\text{dis}(v.cp, u^*.cp) \leq \text{radAt}(v.lev)$, $\text{dis}(v.cp, x) \leq \text{radAt}(v.lev)$, $\text{dis}(u_i.cp, x) \leq \text{radAt}(u_i.lev)$, $u^*.dis2cp$
$= \text{dis}(v.cp, u^*.cp)$, and $u_i.dis2cp = \text{dis}(v.cp, u_i.cp)$.

Case (1). Let LB and UB denote the lower bound and upper bound of distances from $u^*.cp$ to any object z that is in the query range and contained in u^*. Let lb and ub denote the lower bound and upper bound of distance from $u^*.cp$ and x. The proof of Lemma 2 (2) has shown the correctness of LB and $\text{dis}(u^*.cp, z) \leq \text{dis2q}^* + r$. Hence, we only need to prove $\text{dis}(u^*.cp, z) \leq 2\text{radAt}(v.lev)$. As both $u^*.cp$ and z are contained in v, we know $\text{dis}(v.cp, u^*.cp) \leq \text{radAt}(v.lev)$ and $\text{dis}(v.cp, z) \leq \text{radAt}(v.lev)$. By triangle inequality, we have $\text{dis}(v.cp, u^*.cp) + \text{dis}(v.cp, z) \leq \text{radAt}(v.lev) + \text{radAt}(v.lev)$.

We next prove the correctness of lb. By the proof of Lemma 2 (4), we know $\text{dis}(u^*.cp, x)$ is no smaller than $\text{radAt}(v.lev + 1)$ and $\text{disTAt}(v.lev)/v.distort$. Therefore, we can derive $\text{dis}(u^*.cp, x) \geq \text{dis}(u^*.cp, u_i.cp) - \text{dis}(u_i.cp, x) \geq |\text{dis}(v.cp, u^*.cp) - \text{dis}(v.cp, u_i.cp)| - \text{radAt}(u_i.lev) \geq |u^*.dis2cp - u_i.dis2cp| - \text{radAt}(u_i.lev)$.

Then, we prove the correctness of ub. From the proof of Lemma 2 (4), we know $\mathsf{dis}(u^*.cp, x) \leq u^*.dis2cp + \mathsf{radAt}(v.lev)$. Similar to the proof of lb, we can derive that $\mathsf{dis}(u^*.cp, x) \leq \mathsf{dis}(v.cp, u^*.cp) + \mathsf{dis}(v.cp, u_i.cp) + \mathsf{radAt}(u_i.lev) \leq u^*.dis2cp + u_i.dis2cp + \mathsf{radAt}(u_i.lev)$.

We finally prove the statement of the first case. If $\mathsf{ring}(u^*.cp, LB, UB)$ does not overlap with $\mathsf{ring}(u^*.cp, lb, ub)$. the object x cannot be in the query range due to definitions of these bounds.

Case (2). The prerequisite of this case is $\max\{\mathsf{radAt}(v.lev+1), \mathsf{disTAt}(v.lev)/v.distort\} > r + \mathsf{dis2q}^*$, and this case is a corollary of the first case. Let y be an object contained in child nodes $u_i, \cdots, u_k$. Based on lb's definition, we know $lb \geq \max\{\mathsf{radAt}(v.lev+1), \mathsf{disTAt}(v.lev)/v.distort\}$ for any of these child nodes. Based on UB's definition, we know $UB \geq \mathsf{dis2q}^* + r$. Based on the prerequisite, we can infer $lb > UB$, which means $\mathsf{overlap}(u^*.cp, LB, UB, lb, ub)$ is always false for these child nodes. Thus, u_i-u_k can be pruned. □

The proof of **Lemma 4** is as follows.

Proof. Let x be an object contained in u_1 and y be an object contained in v_2-v_k. Based on the node structure and construction algorithm, we know $p.cp = v_1.cp = u_1.cp$, $\mathsf{dis}(p.cp, x) = \mathsf{dis}(u_1.cp, x) \leq \mathsf{radAt}(u_1.lev)$, $\mathsf{dis}(p.cp, y) = \mathsf{dis}(v_1.cp, y)$ is between $\mathsf{radAt}(v_1.lev+1)$ and $\mathsf{radAt}(v_1.lev)]$. Based on the definition of $\mathsf{radAt}(\cdot)$, we know $\mathsf{radAt}(v_1.lev) \geq 2\mathsf{radAt}(u_1.lev) \geq 2^2\mathsf{radAt}(u_1.lev+1)$.

We first derive the lower bound of $\mathsf{dis}(q, y)$ as follows.

$$\begin{aligned}
\mathsf{dis}(q, y) &\geq \mathsf{dis}(p.cp, y) - \mathsf{dis}(p.cp, q) \geq \mathsf{dis}(p.cp, y) - \mathsf{dis2q} \\
&> \mathsf{radAt}(v_1.lev+1) - \mathsf{radAt}(u_1.lev+1) \\
&> 2^2\mathsf{radAt}(u_1.lev+1) - \mathsf{radAt}(u_1.lev+1) \\
&> 3\mathsf{radAt}(u_1.lev+1)
\end{aligned}$$

We next derive the upper bound of $\mathsf{dis}(q, x)$ as follows.

$$\begin{aligned}
\mathsf{dis}(q, x) &\leq \mathsf{dis}(p.cp, x) + \mathsf{dis}(p.cp, q) \leq \mathsf{dis}(p.cp, x) + \mathsf{dis2q} \\
&\leq \mathsf{radAt}(u_1.lev) + \mathsf{radAt}(u_1.lev+1) \\
&\leq 2\mathsf{radAt}(u_1.lev+1) + \mathsf{radAt}(u_1.lev+1) \\
&\leq 3\mathsf{radAt}(u_1.lev+1)
\end{aligned}$$

Based on the prerequisite, u_1 contains at least k objects (*e.g.*, x). The proof above indicates the their distances to q are no longer than $3\mathsf{radAt}(u_1.lev+1)$. Thus, the distance between q and its kth nearest neighbor is no longer than $3\mathsf{radAt}(u_1.lev+1)$. Since $\mathsf{dis}(q, y) > 3\mathsf{radAt}(u_1.lev+1)$, the object y cannot be the k nearest neighbors of q and hence can be pruned. □

5 EXPERIMENTAL STUDY

This section conducts experiments under two scenarios: main-memory and external-memory.

5.1 Experiment in Main-Memory

As aforementioned, in-memory similarity search is our major concern and hence we first conduct experiments in main-memory.

5.1.1 Experimental Setup. **Datasets.** Table 2 lists the datasets used in our experiments. *Color* [37] is an image dataset, where each object is associated with 112-dimensional features extracted from color histograms of 112,682 MPEG-7 images and the L_2 distance is suggested in [37]. *Dutch* [37] and *English* [1] are both string datasets, where each object is a word in Dutch and English, respectively. Thus, we use edit distance to measure the similarity in these two datasets. *BinStr* is a synthetic dataset to test the scalability, where each object is a unique binary string generated at random.

Table 2. Statistics of the datasets

Datasets	#(Objects)	#(Dimension)	Distance function
Color	112,682	112	L_2 distance
Dutch	229,328	1-40	Edit distance
English	466,550	1-45	Edit distance
BinStr	1,000,000	48	Hamming distance

Table 3. Parameter Settings

Parameters	Settings
Radius r	2%, 4%, 8%, 16%, 32% (× the maximum distance)
Integer k	5, 10, 20, 50, 100

Parameters. Table 3 lists the varied parameters in our experiments. Specifically, we vary the length of searching radius r in range queries and increase the value of k in kNN queries. The test ranges of these two parameters are based on the existing survey [26], which are generally aligned with existing studies on similarity search.

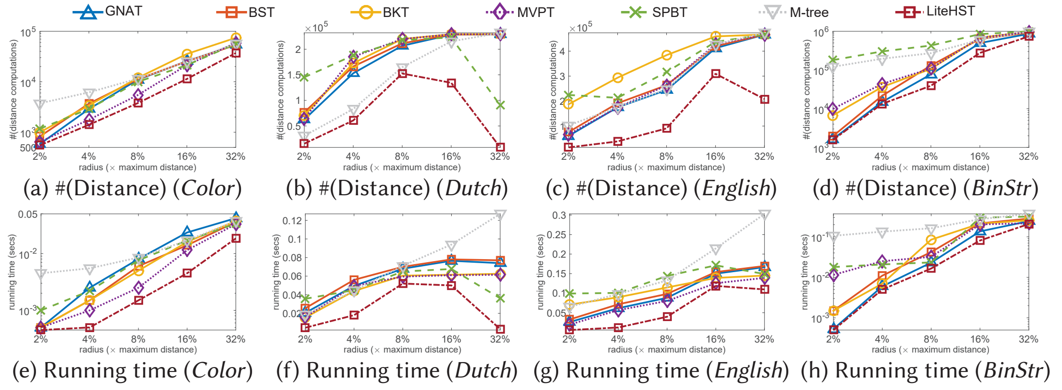
Compared Algorithms. To show the good efficiency of our solution (denoted by LiteHST), our experiments compare with the state-of-the-art solutions from two aspects: query efficiency by different indexes and query efficiency by different processing methods over same index (*i.e.*, our LiteHST), in Sec. 5.1.2 and Sec. 5.1.3.

First, we want to show the superior performance of our index LiteHST. Thus, based on a recent survey [26] for in-memory similarity search, we choose the top 4 fastest indexes as our baseline indexes, *i.e.*, MVPT [16, 17], GNAT [18], BST [59] and BKT [20]. These indexes are also devised based on either partitioning or embedding. We also compare with in-memory M-tree [3, 27] and the embedding-based index SPB-tree (SPBT for short) in a recent work [25]. As SPBT is a B-tree based index, we also use the learned index PGM-index [36] to enhance its query efficiency and space cost. All embedding-based indexes adopt the pivot selection algorithm HFI [25], since it often leads to the best query efficiency [84].

Besides, we also want to demonstrate that simple extensions of existing query processing methods cannot fully utilize the structure properties of LiteHST and hence sometimes have worse efficiency than the best among existing indexes. In this part, we have the following baselines from existing literature.

- LiteHST_{Mtree} answers the similarity search over LiteHST by using the query processing of the M-tree family [27], since each node in M-tree also corresponds to a ball partition.
- $\text{LiteHST}_{FastMap}$ solves the similarity search over LiteHST by using a general framework [47] for contractive embeddings.
- SOTA denotes the best result achieved by existing indexes, *i.e.*, MVPT, GNAT, BST, BKT, SPBT, and M-tree.
- LiteHST_{O0} denotes the result of kNN query over LiteHST without our learning-based optimization.

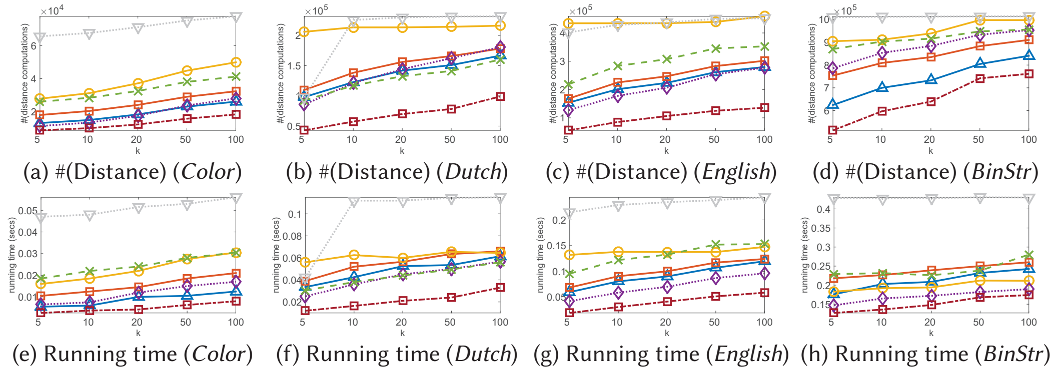
Implementation. All the algorithms are coded in C++ with a uniform implementation, and most implementations of the existing solutions are referred to the well-known and open-sourced library [37] maintained by the International Workshop on Similarity Search and Applications (SISAP). For the learning-based optimization in kNN query, we use the Gradient Boosting Regression Tree (GBRT) [45] implemented in scikit-learn [2] as our regression models. Similar to other learning-enhanced index work [36, 61, 66], regression models have been trained in advance of queries.



(a) #(Distance) (*Color*) (b) #(Distance) (*Dutch*) (c) #(Distance) (*English*) (d) #(Distance) (*BinStr*)

(e) Running time (*Color*) (f) Running time (*Dutch*) (g) Running time (*English*) (h) Running time (*BinStr*)

Fig. 3. Comparisons with GNAT, BST, BKT, MVPT, SPBT, and M-tree in range queries



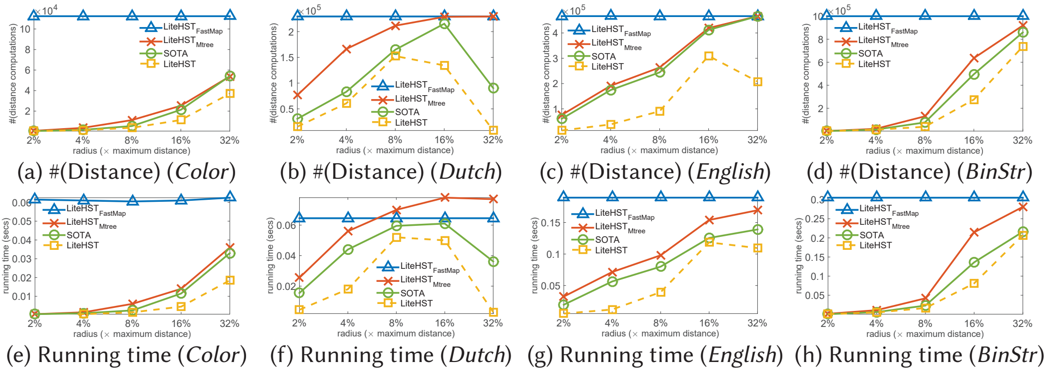
(a) #(Distance) (*Color*) (b) #(Distance) (*Dutch*) (c) #(Distance) (*English*) (d) #(Distance) (*BinStr*)

(e) Running time (*Color*) (f) Running time (*Dutch*) (g) Running time (*English*) (h) Running time (*BinStr*)

Fig. 4. Comparisons with GNAT, BST, BKT, MVPT, SPBT, and M-tree in kNN queries

Metrics. These algorithms are evaluated in terms of *the number of distance computations* ("#(Distance)" for short) and *the running time*, which are common metrics in existing work. We also report the results of *running time* and *space consumption* on constructing these indexes. All experiments are conducted on a server with the Intel Xeon(R) Gold 6240R 2.40GHz processor, 128 GB RAM and 1 TB disk space. Each parameter setting is repeated 50 times with randomly sampled query objects, and the average result is reported.

5.1.2 Comparisons with Existing Indexes. **Results of Range Query.** Fig. 3 illustrates the experimental comparisons with existing indexes in range queries. We can observe that our LiteHST is always the most efficient in all four datasets. Specifically, LiteHST saves up to 1.8×, 11.0×, 4.5× and 1.9× fewer distance computations than the runner-up in *Color*, *Dutch*, *English*, and *BinStr* datasets, respectively. This time saving is due to the effective pruning rules designed in Lemmas 2 and 3. As a result, LiteHST is up to 19.5×-211.0× faster than the compared baselines in these datasets. Besides, we can also observe that the efficiency of existing baselines generally gets worse with the expansion of query ranges. However, for our index LiteHST, when the searching radius becomes large enough, the efficiency sometimes gets better (*e.g.*, Fig. 3f and Fig. 3g). This is due to the pruning rules of Lemma 2 (1) and (3). That is, when the ball areas of more nodes get fully covered by the increasingly enlarged query range, we do not have to check the distances from the objects in these nodes to the query object, which reduces the time cost. Among the baselines, MVPT is the fastest in *Color* and *English* datasets, GNAT is often the most efficient in *BinStr* dataset,



(a) #(Distance) (*Color*) (b) #(Distance) (*Dutch*) (c) #(Distance) (*English*) (d) #(Distance) (*BinStr*)

(e) Running time (*Color*) (f) Running time (*Dutch*) (g) Running time (*English*) (h) Running time (*BinStr*)

Fig. 5. Comparisons with LiteHST$_{FastMap}$, LiteHST$_{Mtree}$ and SOTA in range queries

Table 4. Results of construction in *BinStr*

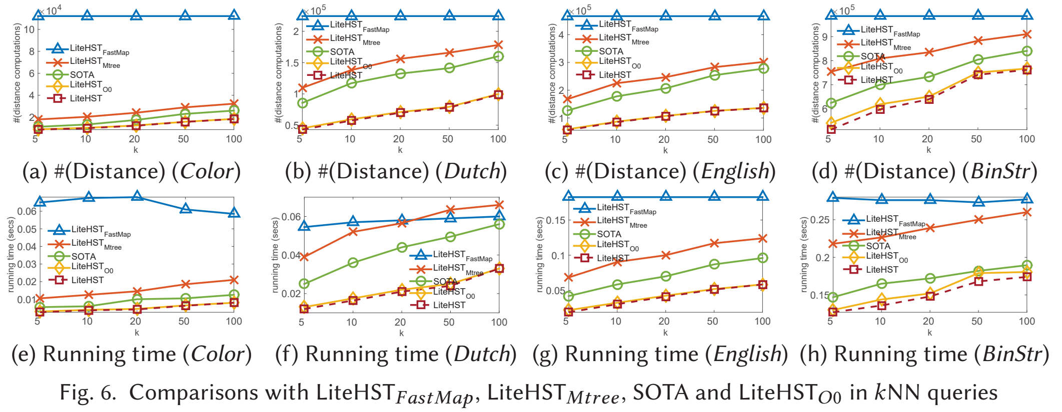
Index	BST	BKT	GNAT	MVPT	SPBT	M-tree	Ours
Time	4.9s	1.5s	2.0s	1.6s	26.6s	18.3s	86.5s
Space	27M	9M	31M	7M	23M	43M	34M

BKT and SPBT are sometimes the fastest in *Dutch* dataset, and M-tree sometimes has the least distance computations in *Dutch* dataset.

Results of kNN Query. Fig. 4 presents the experimental comparisons with existing indexes in kNN queries. Our solution LiteHST always takes the least distance computations and the shortest running time. For example, LiteHST is 1.3×-17.0× faster than the compared baselines in the *Color* dataset, and it needs 2.2×-7.7× fewer distance computations in the *English* dataset. Besides, we also observe the running time of all the solutions gets longer with the increase of k. Among these baselines, GNAT is the fastest in *Color* datasets, and MVPT is the fastest in *English* and *BinStr* datasets. Although BKT or SPBT is often the least efficient, they are sometimes faster than GNAT and BST. M-tree is often the slowest baseline. The results also validate our motivation that no single method dominates all the others.

Results of Construction. Due to page limitations, we only list the results of construction in the large-scale dataset (*i.e.*,*BinStr*). As shown in Table 4, all these methods can construct an index of up to 1 million objects in less than 1.5 minutes with no more than 43 MB space, which is relatively efficient for real-world scenarios. Among these compared baselines, BKT takes the lowest time cost and MVPT consumes the least space cost. The results of our LiteHST are consistent with the time and space complexity of the LiteHST, *i.e.*, $O(dn \log n)$ and $O(n)$ respectively, where $O(d)$ is the time cost of a distance computation and n is the number of objects. Although the results show that constructing LiteHST takes longer time and more space than the others, this is still acceptable when considering the notable improvement in query efficiency.

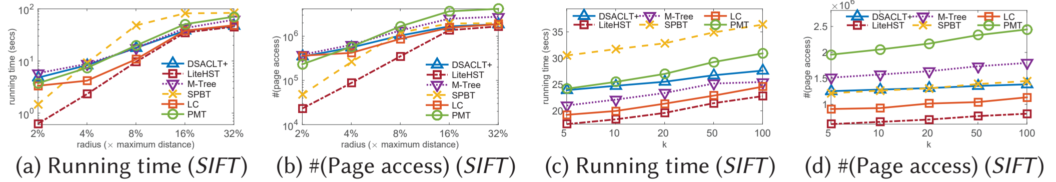
5.1.3 Comparisons with Existing Query Processing Methods Over LiteHST. **Results of Range Query.** Fig. 5 presents the experimental comparisons with existing query processing methods over LiteHST on range queries. First, we can observe that the state-of-the-art index (*i.e.*, SOTA) is not outperformed by existing approaches over LiteHST, *i.e.*, LiteHST$_{FastMap}$ and LiteHST$_{Mtree}$. However, our query processing algorithm LiteHST is always faster than SOTA. Second, we can also observe that the efficiency of LiteHST$_{FastMap}$ is relatively stable. This is because the pruning rule of LiteHST$_{FastMap}$ relies on the value of distortion, which takes $O(n^2)$ time in the construction and hence is too slow for an in-memory index for similarity search. As a result, we use the theoretical



(a) #(Distance) (*Color*) (b) #(Distance) (*Dutch*) (c) #(Distance) (*English*) (d) #(Distance) (*BinStr*)

(e) Running time (*Color*) (f) Running time (*Dutch*) (g) Running time (*English*) (h) Running time (*BinStr*)

Fig. 6. Comparisons with LiteHST$_{FastMap}$, LiteHST$_{Mtree}$, SOTA and LiteHST$_{O0}$ in kNN queries



(a) Running time (*SIFT*) (b) #(Page access) (*SIFT*) (c) Running time (*SIFT*) (d) #(Page access) (*SIFT*)

Fig. 7. Comparisons with DSACLT+, LC, PMT, SPBT, and M-tree in range queries (a-b) and kNN queries (c-d)

guarantee of the distortion (*i.e.*, $96 \log n$ in Lemma 1), which is too loose in practice. These issues cause LiteHST$_{FastMap}$ to be as slow as a linear scan since its pruning rule is often unsatisfactory. Finally, these results validate that our solution utilizes the structure properties of LiteHST better than the existing solutions.

Results of kNN Query. Fig. 6 depicts the experimental comparisons with existing query processing methods over LiteHST on kNN queries. Overall, we can observe a similar pattern with the experiments on range queries. That is, SOTA is still faster than LiteHST$_{FastMap}$ and LiteHST$_{Mtree}$, but slower than our query processing algorithm LiteHST in all these tests. Moreover, we can also evaluate the effect of our learning-based optimization by comparing between LiteHST and LiteHST$_{O0}$. The comparison shows that our learning-based enhancement can save up to 5.52% of the number of distance computations and reduce the time cost by up to 6.55%.

5.1.4 Summary of Major Experimental Findings. We have following observations in our experiments under the main-memory scenario.

(1) The experiments in Sec. 5.1.2 validate our motivation, *i.e.*, *no single solution can dominate other existing approaches in query efficiency of similarity search*. For example, MVPT and GNAT are often more efficient than BST and BKT. Although BKT is often the least efficient, sometimes it can be faster than GNAT and BST. M-tree can have the least distance computations among baselines.

(2) The experiments in Sec. 5.1.3 prove that it is challenging and non-trivial to devise query processing methods for tree embeddings.

(3) Our LiteHST outperforms the state-of-the-art solutions in query efficiency in two aspects: number of distance computations and running time. For example, LiteHST is up to 19.5×-211.0× faster than MVPT, GNAT, BST, BKT, SPBT, and M-tree in range queries, and up to 1.8×-17.1× faster than them in kNN queries.

5.2 Experiment in External-Memory

We also conduct an experiment in the external-memory scenario, since the dataset may not fit the size of the main-memory.

5.2.1 Experimental Setup. **Datasets.** We use a public dataset called *SIFT* [4] with 100 million objectives on the L_2 metric space. Each objective is associated with 128-dimensional features. The parameter settings for range queries and *k*NN queries are listed in Table 3.

Compared Algorithms. According to the recent survey [26], we choose the (averagely) top 4 fastest indexes for *external-memory similarity search* as our baselines, *i.e.*, DSACLT+ [19], LC [39], PM-tree [72] (PMT for short), and (external-memory) SPB-tree (SPBT for short) [25]. We also compare with (external-memory) M-tree [27]. Note that since PGM-index [36] is an in-memory learned index to enhance the performance of (in-memory) SPBT, we can no longer adopt this learning-based optimization here.

Implementation. All solutions are coded in C++ with a uniform implementation based on the open-sourced library [37] maintained by SISAP. All embedding-based indexes adopt the same pivot selection algorithm called HFI [25] to pick 5 pivots, which often leads to the best query efficiency based on the evaluation work [84]. They are configured to use a fixed disk page size of 4KB and a cache size of 256KB. These configurations are commonly seen in existing research [9, 26, 33, 67]. The bucket size of LC is configured to be 1024KB, which can be regarded as 256 disk pages. This parameter has been tuned by us, since a smaller bucket indicates (almost) unacceptable construction time (*e.g.*, over 2 days), and a larger size indicates a worse query efficiency. Other implementation details are similar to those in Sec. 5.1.1.

Metrics. These solutions are evaluated in terms of *the number of page access* ("#(Page access)" for short) and *the running time* (including both CPU time and I/O time). We report the number of page accesses instead of the distance computations, since the former is more important to the running time than the latter in the external-memory scenario. We also report the results of *construction time* and *index size*. Each parameter setting is repeated 50 times with random objects, and the average result is reported. The experimental environment is the same as that in Sec. 5.1.1. All these experiments are conducted on a server with the Intel Xeon(R) Gold 6240R 2.40GHz processor, 128 GB RAM and 1 TB external disk space.

5.2.2 Experimental Results. **Results of Range Query.** Fig. 7a and Fig. 7b present the experimental comparisons with existing external-memory indexes for range queries. We can first observe that the running time and the number of page accesses will increase when the radius gets longer. This is because more objects will be involved query answers when the query range gets larger. Our LiteHST is always the most efficient in terms of both running time and I/O cost (*i.e.*, the number of page accesses). For example, LiteHST is up to 4.0×, 4.3×, 5.0×, 6.6×, and 8.3× faster than SPBT, LC, PMT, DSACLT+, and M-tree, respectively. The number of page accesses by LiteHST is up to 3.5×-16.4× smaller than those by compared baselines. Besides, when the radius is the longest, the top 3 fastest algorithms are LiteHST, DSACLT+, and LC (from the fastest solution to the slowest one). They have closer running time and similar numbers of page accesses, because large portions of objects are covered by the largest query range and large numbers of disk pages are accessed. Overall, the improvements of LiteHST are due to our well-structured index and pruning rules in the query processing. These results demonstrate that our solution LiteHST can still be a promising solution in the external-memory scenario. Among these baselines, LC is often the most efficient. SPBT and DSACLT+ can sometimes be the most efficient. PMT often has the largest number of page accesses. Sometimes M-tree takes the longest running time. The results also show that no single method always dominates all the others in the external-memory scenario.

Table 5. Results of construction in *SIFT* (time: minute)

Index	DSACLT+	LC	PMT	SPBT	M-tree	Ours
Time	9.5	2986	339	293	52	354
Space	162M	5M	553M	231M	215M	522M

Results of *k*NN Query. Fig. 7c and Fig. 7d illustrate the experimental results of *k*NN queries in the external-memory scenario. In these results, it is obvious that our solution LiteHST still takes the shortest running time and the least number of page accesses. For instance, the number of page accesses by our solution is up to 1.5×-3.2× smaller than those by compared baselines, which leads to the decrease in the time cost of LiteHST. In terms of running time, the rankings of the baselines are LC, M-tree, DSACLT+, PMT, and SPBT (from the fastest baseline to the slowest one). As for the number of page accesses, LC is still the best among these baselines, and DSACLT+ or SPBT can sometimes be the runner-up. PMT has the highest I/O cost, since it takes the largest number of page accesses.
Results of Construction. As shown in Table 5, most of these methods can construct an index of 100 million objects in less than 6 hours with no more than 553 MB space. Our method is comparably fast with PMT, and faster than LC. Among these compared baselines, DSACLT+ is the fastest, and M-tree is the runner-up in the time cost. LC, which takes over 2 days, is the slowest, but takes the least space cost. Overall, although LiteHST takes a longer time and more space than some of the baselines, this is acceptable when considering the data size and improvement in query efficiency.

5.2.3 Summary of Major Experimental Findings. We have the following observations in our experiment in the external-memory.

(1) Our solution LiteHST can also efficiently process datasets that do not fit the size of the main-memory. Moreover, the running time of LiteHST is notably shorter than the compared baselines in both range queries and *k*NN queries.

(2) Among the baselines, LC is often the most efficient for both range queries and *k*NN queries. However, the construction time of LC is much longer than the others. M-tree, DSACLT+, and SPBT are relatively efficient. For example, SPBT is sometimes the fastest baseline in range queries. M-tree is the runner-up baseline in terms of running time in *k*NN queries. DSACLT+ sometimes has fewer numbers of page accesses than M-tree and PMT.

6 RELATED WORK

Our paper is related to *similarity search* and *tree embeddings*.
Similarity Search. Similarly search has been widely studied in the Database community. Following the taxonomy in [26], we briefly review the existing research from two categories: *partitioning based solutions* and *pivot based solutions*. Please refer to the surveys [23, 26, 46, 69] and books [71, 82] for a more detailed review.

The basic idea of *partitioning based solutions* is to first separate the objects into disjoint partitions and then prune the dissimilar objects in specific partitions. The compared algorithms in our experiments, BST [59] and M-tree [27], belong to this category. Other examples include Spatial Approximation Tree [64], LC family [22], D-index family [30, 31], and Δ-tree [29]. Our proposed index LiteHST also applies a ball partition based technique.

For the *pivot based solutions*, the basic idea is to utilize the distances to a few selected pivots and the properties like the symmetry and triangle inequality to derive the distance bounds and refine the search space. Here, a pivot is a reference object that is often selected from the input objects V and pivot mapping aims to embed the input metric into an L_p metric via these pivots. Generally

speaking, pivot mapping is a special version of Lipschitz Embedding (defined in Sec. 2.2), where the cardinality of each reference set is 1. Examples in this category include GNAT [18], BKT [20], MVPT [16, 17], Omni-family [58] and SPB-tree [25].

Recently, existing research also focuses on approximation solutions to similarity search (*i.e.*, "*approximate similarity search*"). A few popular techniques for approximate similarity search include locality-sensitive hashing (LSH) [43] (*e.g.*, QALSH [55] and PM-LSH [83]), permutation-based indexes [38, 77], pivot-based pruning (*e.g.*, HD-index [9]), local intrinsic dimensionality (LID) based optimization [52, 53], learning-based enhancement [8], and so on. Please refer to the tutorials [32, 68] and benchmarks [10] for recent techniques for approximate similarity search. Note that these algorithms usually find approximate answers instead of exact answers, or can only retrieve exact answers with certain probabilities. By contrast, based on our problem definition in Def. 2-3, our solution and compared baselines need to always find the exact answers. Thus, these approximation solutions were not compared in our experiments.

Tree Embeddings. Bartal [12] proposed the first work on tree embeddings, *i.e.*, Hierarchically Separated Tree (HST). From then on, many studies have been proposed to find good tree embeddings.

The basic idea of tree embeddings is to map the objects in arbitrary metric spaces into a tree. For example, we can map the objects in V into the leaves of an HST, where HST is the tree embedding of the input metric (V, dis). Early work [13, 35, 60] concentrated on studying the distortion guarantee of a tree embedding, where the distortion is the standard measurement of the quality of an embedding. Recent studies focus on efficiently constructing a tree embedding [14, 40, 42, 80, 81]. or broadening the applications of tree embeddings (*e.g.*, clustering [11], privacy protection [24, 73], facility location [34], spatial crowdsourcing [74, 75], and route planning [28, 79]). To the best of our knowledge, tree embeddings have never been used for exact similarity search.

7 CONCLUSION

This paper focuses on the in-memory similarity search, where similarity is measured by arbitrary metrics and no feature vectors are known in advance to represent the objects. There is still no single solution that can dominate all the other methods in terms of the query efficiency. To achieve this goal, we are motivated by a new embedding technique for similarity search: tree embeddings. Specifically, we propose a new index called LiteHST based on the classic tree embedding, HST. Based on this new index, we propose efficient query processing methods with complex pruning strategies for range and kNN queries in similarity search. Moreover, a learning-based optimization is used to accelerate kNN queries. Finally, extensive experiments demonstrate that our solution LiteHST outperforms the state-of-the-art solutions in the query efficiency.

ACKNOWLEDGMENTS

This work is partially supported by National Key Research and Development Program of China Grant No. 2018AAA0101100, National Science Foundation of China (NSFC) under Grant No. U22B2060, the Hong Kong RGC GRF Project 16209519, CRF Project C6030-18G, C2004-21GF, AOE Project AoE/E-603/18, RIF Project R6020-19, Theme-based project TRS T41-603/20R, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants MHX/078/21 and PRP/004/22FX, Microsoft Research Asia Collaborative Research Grant, HKUST-Webank joint research lab grant and HKUST Global Strategic Partnership Fund (2021 SJTU-HKUST). Yongxin Tong's work is partially supported by National Key Research and Development Program of China Grant No. 2018AAA0101100, National Science Foundation of China (NSFC) under Grant No. U21A20516 and 62076017, the Beihang University Basic Research Funding No. YWF-22-L-531, the Funding No. 22-TQ23-14-ZD-01-001 and WeBank Scholars Program.

REFERENCES

[1] 2021. List of English words. https://github.com/dwyl/english-words/
[2] 2021. Scikit-learn. https://scikit-learn.org/stable/
[3] 2022. The in-memory M-tree. https://github.com/erdavila/M-Tree
[4] 2022. The SIFT dataset. http://corpus-texmex.irisa.fr/
[5] Ittai Abraham, Yair Bartal, and Ofer Neiman. 2006. Advances in metric embedding theory. In *STOC*. 271–286.
[6] Laurent Amsaleg, Oussama Chelly, Teddy Furon, Stéphane Girard, Michael E. Houle, Ken-ichi Kawarabayashi, and Michael Nett. 2015. Estimating Local Intrinsic Dimensionality. In *SIGKDD*. 29–38.
[7] Laurent Amsaleg, Oussama Chelly, Michael E. Houle, Ken-ichi Kawarabayashi, Milos Radovanovic, and Weeris Treeratanajaru. 2019. Intrinsic Dimensionality Estimation within Tight Localities. In *SDM*. 181–189.
[8] Matej Antol, Jaroslav Olha, Terézia Slanináková, and Vlastislav Dohnal. 2021. Learned Metric Index - Proposition of learned indexing for unstructured data. *Inf. Syst.* 100 (2021), 101774.
[9] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. 2018. HD-Index: Pushing the Scalability-Accuracy Boundary for Approximate kNN Search in High-Dimensional Spaces. *PVLDB* 11, 8 (2018), 906–919.
[10] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.
[11] Arturs Backurs, Piotr Indyk, Krzysztof Onak, Baruch Schieber, Ali Vakilian, and Tal Wagner. 2019. Scalable Fair Clustering. In *ICML*. 405–413.
[12] Yair Bartal. 1996. Probabilistic Approximations of Metric Spaces and Its Algorithmic Applications. In *FOCS*. 184–193.
[13] Yair Bartal. 1998. On Approximating Arbitrary Metrices by Tree Metrics. In *STOC*. 161–168.
[14] Guy E. Blelloch, Anupam Gupta, and Kanat Tangwongsan. 2012. Parallel probabilistic tree embeddings, k-median, and buy-at-bulk network design. In *SPAA*. 205–213.
[15] Jean Bourgain. 1985. On Lipschitz embedding of finite metric spaces in Hilbert space. *Israel Journal of Mathematics* 52 (1985), 46–52.
[16] Tolga Bozkaya and Z. Meral Özsoyoglu. 1997. Distance-Based Indexing for High-Dimensional Metric Spaces. In *SIGMOD*. 357–368.
[17] Tolga Bozkaya and Z. Meral Özsoyoglu. 1999. Indexing Large Metric Spaces for Similarity Search Queries. *ACM Trans. Database Syst.* 24, 3 (1999), 361–404.
[18] Sergey Brin. 1995. Near Neighbor Search in Large Metric Spaces. In *VLDB*. 574–584.
[19] Luis Britos, A. Marcela Printista, and Nora Reyes. 2012. DSACL+-tree: A Dynamic Data Structure for Similarity Search in Secondary Memory. In *SISAP*. 116–131.
[20] Walter A. Burkhard and Robert M. Keller. 1973. Some Approaches to Best-Match File Searching. *Commun. ACM* 16, 4 (1973), 230–236.
[21] Guillaume Casanova, Elias Englmeier, Michael E. Houle, Peer Kröger, Michael Nett, Erich Schubert, and Arthur Zimek. 2017. Dimensional Testing for Reverse K-Nearest Neighbor Search. *PVLDB* 10, 7 (2017), 769–780.
[22] Edgar Chávez and Gonzalo Navarro. 2000. An Effective Clustering Algorithm to Index High Dimensional Metric Spaces. In *SPIRE*. 75–86.
[23] Edgar Chávez, Gonzalo Navarro, Ricardo A. Baeza-Yates, and José L. Marroquín. 2001. Searching in metric spaces. *ACM Comput. Surv.* 33, 3 (2001), 273–321.
[24] Shuchi Chawla, Cynthia Dwork, Frank McSherry, and Kunal Talwar. 2005. On Privacy-Preserving Histograms. In *UAI*. 120–127.
[25] Lu Chen, Yunjun Gao, Xinhan Li, Christian S. Jensen, and Gang Chen. 2017. Efficient Metric Indexing for Similarity Search and Similarity Joins. *IEEE Trans. Knowl. Data Eng.* 29, 3 (2017), 556–571.
[26] Lu Chen, Yunjun Gao, Xuan Song, Zheng Li, Yifan Zhu, Xiaoye Miao, and Christian S. Jensen. 2023. Indexing Metric Spaces for Exact Similarity Search. *ACM Comput. Surv.* 55, 6 (2023), 128:1–128:39.
[27] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *PVLDB*. 426–435.
[28] Christian Coester and Elias Koutsoupias. 2019. The online k-taxi problem.. In *STOC*. 1136–1147.
[29] Bin Cui, Beng Chin Ooi, Jianwen Su, and Kian-Lee Tan. 2005. Indexing High-Dimensional Data for Efficient In-Memory Similarity Search. *IEEE Trans. Knowl. Data Eng.* 17, 3 (2005), 339–353.
[30] Vlastislav Dohnal. 2004. An Access Structure for Similarity Search in Metric Spaces. In *EDBT*. 133–143.
[31] Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. 2003. D-Index: Distance Searching Index for Metric Data Sets. *Multim. Tools Appl.* 21, 1 (2003), 9–33.
[32] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2021. High-Dimensional Similarity Search for Scalable Data Science. In *ICDE*. 2369–2372.
[33] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2018. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB* 12, 2 (2018), 112–127.

[34] Yunus Esencayi, Marco Gaboardi, Shi Li, and Di Wang. 2019. Facility Location Problem in Differential Privacy Model Revisited. In *NeurIPS*. 8489–8498.

[35] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. 2003. A tight bound on approximating arbitrary metrics by tree metrics. In *STOC*. 448–455.

[36] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB* 13, 8 (2020), 1162–1175.

[37] Karina Figueroa, Gonzalo Navarro, and Edgar Chavez. 2017. The Metric Spaces Library maintained by the SISAP initiative. https://github.com/kaarinita/metricSpaces

[38] Karina Figueroa and Nora Reyes. 2019. Permutation's Signatures for Proximity Searching in Metric Spaces. In *SISAP*. 151–159.

[39] Kimmo Fredriksson. 2007. Engineering efficient metric indexes. *Pattern Recognit. Lett.* 28, 1 (2007), 75–84.

[40] Stephan Friedrichs and Christoph Lenzen. 2018. Parallel Metric Tree Embedding Based on an Algebraic View on Moore-Bellman-Ford. *J. ACM* 65, 6 (2018), 43:1–43:55.

[41] Keinosuke Fukunaga. 2013. *Introduction to statistical pattern recognition.* Elsevier.

[42] Jie Gao, Leonidas J. Guibas, Nikola Milosavljevic, and Dengpan Zhou. 2009. Distributed resource management and matching in sensor networks. In *IPSN*. 97–108.

[43] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*. 518–529.

[44] Sariel Har-Peled. 2011. *Geometric approximation algorithms.* American Mathematical Society.

[45] Trevor Hastie, Jerome H. Friedman, and Robert Tibshirani. 2001. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer.

[46] Gísli R. Hjaltason and Hanan Samet. 2003. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.* 28, 4 (2003), 517–580.

[47] Gísli R. Hjaltason and Hanan Samet. 2003. Properties of Embedding Methods for Similarity Searching in Metric Spaces. *IEEE Trans. Pattern Anal. Mach. Intell.* 25, 5 (2003), 530–549.

[48] Michael E. Houle. 2013. Dimensionality, Discriminability, Density and Distance Distributions. In *2013 IEEE 13th International Conference on Data Mining Workshops*. 468–473.

[49] Michael E. Houle. 2017. Local Intrinsic Dimensionality I: An Extreme-Value-Theoretic Foundation for Similarity Applications. In *SISAP*. 64–79.

[50] Michael E. Houle. 2017. Local Intrinsic Dimensionality II: Multivariate Analysis and Distributional Support. In *SISAP*. 80–95.

[51] Michael E. Houle. 2020. Local Intrinsic Dimensionality III: Density and Similarity. In *SISAP*. 248–260.

[52] Michael E. Houle, Vincent Oria, Kurt R. Rohloff, and Arwa M. Wali. 2018. LID-Fingerprint: A Local Intrinsic Dimensionality-Based Fingerprinting Method. In *SISAP*. 134–147.

[53] Michael E. Houle, Vincent Oria, and Arwa M. Wali. 2017. Improving k-NN Graph Accuracy Using Local Intrinsic Dimensionality. In *SISAP*. 110–124.

[54] Gabriela Hristescu and Martin Farach-Colton. 1999. *Cluster-preserving embedding of proteins.* Technical Report. Computer Science Department, Rutgers University.

[55] Qiang Huang, Jianlin Feng, Qiong Fang, Wilfred Ng, and Wei Wang. 2017. Query-aware locality-sensitive hashing scheme for L_p norm. *VLDB J.* 26, 5 (2017), 683–708.

[56] Piotr Indyk. 2001. Algorithmic Applications of Low-Distortion Geometric Embeddings. In *FOCS*. 10–33.

[57] William B Johnson and Joram Lindenstrauss. 1984. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary mathematics* 26 (1984), 189–206.

[58] Caetano Traina Jr., Roberto F. Santos Filho, Agma J. M. Traina, Marcos R. Vieira, and Christos Faloutsos. 2007. The Omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient. *VLDB J.* 16, 4 (2007), 483–505.

[59] Iraj Kalantari and Gerard McDonald. 1983. A Data Structure and an Algorithm for the Nearest Point Problem. *IEEE Trans. Software Eng.* 9, 5 (1983), 631–634.

[60] Goran Konjevod, R. Ravi, and F. Sibel Salman. 2001. On approximating planar metrics by tree metrics. *Information Processing Letters* 80, 4 (2001), 213–219.

[61] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504.

[62] Nathan Linial, Eran London, and Yuri Rabinovich. 1994. The geometry of graphs and some of its algorithmic applications. In *FOCS*. 577–591.

[63] Jirí Matousek. 2002. *Lectures on discrete geometry.* Graduate texts in mathematics, Vol. 212. Springer.

[64] Gonzalo Navarro. 2002. Searching in metric spaces by spatial approximation. *VLDB J.* 11, 1 (2002), 28–46.

[65] David Novak, Michal Batko, and Pavel Zezula. 2011. Metric Index: An efficient and scalable solution for precise and approximate similarity search. *Inf. Syst.* 36, 4 (2011), 721–733.
[66] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *PVLDB* 13, 11 (2020), 2341–2354.
[67] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2020. Packing R-trees with Space-filling Curves: Theoretical Optimality, Empirical Efficiency, and Bulk-loading Parallelizability. *ACM Trans. Database Syst.* 45, 3 (2020), 14:1–14:47.
[68] Jianbin Qin, Wei Wang, Chuan Xiao, and Ying Zhang. 2020. Similarity Query Processing for High-Dimensional Data. *PVDLB* 13, 12 (2020), 3437–3440.
[69] Dmitri A Rachkovskij. 2017. Distance-Based Index Structures for Fast Similarity Search. *Cybernetics & Systems Analysis* 53, 4 (2017), 636–658.
[70] Richard A Roberts and Clifford T Mullis. 1987. *Digital signal processing*. Addison-Wesley Longman Publishing Co., Inc.
[71] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures*. Academic Press.
[72] Tomás Skopal, Jaroslav Pokorný, and Václav Snásel. 2004. PM-tree: Pivoting Metric Tree for Similarity Search in Multimedia Databases. In *ADBIS*. 803–815.
[73] Qian Tao, Yongxin Tong, Zimu Zhou, Yexuan Shi, Lei Chen, and Ke Xu. 2020. Differentially Private Online Task Assignment in Spatial Crowdsourcing: A Tree-based Approach. In *ICDE*. 517–528.
[74] Yongxin Tong, Jieying She, Bolin Ding, Lei Chen, Tianyu Wo, and Ke Xu. 2016. Online Minimum Matching in Real-Time Spatial Data: Experiments and Analysis. *PVLDB* 9, 12 (2016), 1053–1064.
[75] Yongxin Tong, Zimu Zhou, Yuxiang Zeng, Lei Chen, and Cyrus Shahabi. 2020. Spatial crowdsourcing: a survey. *The VLDB Journal* 29, 1 (2020), 217–250.
[76] Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman. 2017. *Handbook of discrete and computational geometry*. Chapman and Hall/CRC.
[77] Lucia Vadicamo, Richard Connor, Fabrizio Falchi, Claudio Gennaro, and Fausto Rabitti. 2019. SPLX-Perm: A Novel Permutation-Based Representation for Approximate Metric Search. In *SISAP*. 40–48.
[78] David P Williamson and David B Shmoys. 2011. *The design of approximation algorithms*. Cambridge university press.
[79] Yuxiang Zeng, Yongxin Tong, and Lei Chen. 2019. Last-Mile Delivery Made Practical: An Efficient Route Planning Framework with Theoretical Guarantees. *PVLDB* 13, 3 (2019), 320–333.
[80] Yuxiang Zeng, Yongxin Tong, and Lei Chen. 2021. HST+: An Efficient Index for Embedding Arbitrary Metric Spaces. In *ICDE*. 648–659.
[81] Yuxiang Zeng, Yongxin Tong, and Lei Chen. 2022. Faster and Better Solution to Embed Lp Metrics by Tree Metrics. In *SIGMOD*. 2135–2148.
[82] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. 2006. *Similarity Search - The Metric Space Approach*. Advances in Database Systems, Vol. 32. Kluwer.
[83] Bolong Zheng, Xi Zhao, Lianggui Weng, Quoc Viet Hung Nguyen, Hang Liu, and Christian S. Jensen. 2021. PM-LSH: a fast and accurate in-memory framework for high-dimensional approximate NN and closest pair search. *VLDB J.* (2021), 1–25.
[84] Yifan Zhu, Lu Chen, Yunjun Gao, and Christian S. Jensen. 2022. Pivot selection algorithms in metric spaces: a survey and experimental study. *VLDB J.* 31, 1 (2022), 23–47.

Received April 2022; revised July 2022; accepted August 2022