HST+: An Efficient Index for Embedding Arbitrary Metric Spaces

Yuxiang Zeng[†], Yongxin Tong[‡], Lei Chen[†]
[†]The Hong Kong University of Science and Technology, Hong Kong SAR, China
[‡]SKLSDE Lab, BDBC and IRI, Beihang University, China
[†]{yzengal, leichen}@cse.ust.hk, [‡]yxtong@buaa.edu.cn

***Abstract*— Metric embeddings have been widely used in approximate algorithms to guarantee the effectiveness of geometric problems. Among the metric embedding techniques, Hierarchically Separated Tree (HST) is one of the most prevalent data structures, which maps the points of the original metric space into a tree-based metric space. A few selected applications of the HST include clustering, task assignment, trip planning, privacy preservation, information routing in wireless sensor networks, etc. Despite the popularity in ensuring the effectiveness, the HST-based solutions can be *inefficient* in large-scale datasets, since the state-of-the-art construction method has high time and space complexity ($O(n^3)$ and $O(n^2)$ in the worst-case). Moreover, existing studies overlook the insertion of new points in real applications (deletions can be trivially supported), which can cause the reconstruction of the whole HST. To address these limitations, we focus on designing an efficient index for embedding arbitrary metric spaces by tree metric spaces. Specifically, for *construction*, we design a dynamic programming-based method, which significantly reduces the time and space complexity to $O(n^2)$ and $O(n)$ respectively. For *insertion* of new points, we propose a new data structure, called Hierarchically Separated Forest (HSF), *i.e.*, a collection of HSTs. An HSF can efficiently support insertion of new points with a tight theoretical guarantee ($O(\log n)$). Finally, extensive experiments demonstrate the superior performance of our proposed algorithms with respect to the effectiveness and the running time. For instance, compared with the state-of-the-art algorithms, our construction method is up to $29.8\times$ faster and our insertion method is up to $491\times$ faster.**

***Index Terms*—metric embedding, hierarchically separated tree**

I. Introduction

With the development of GPS and mobile devices, geometric data management [1], [2] and geometric applications [3] become increasingly important. In the real-world applications, some of the fundamental problems (*e.g.*, trip planning, task assignment, facility location planning) are hard (*e.g.*, NP-hard problems or online problems), where efficient approximate solutions are often more desired than inefficient exact solutions. To design geometric approximate algorithms, *metric embedding* has been a simple but powerful technique [4]–[6].

Specifically, by metric embedding, one maps the geometric data from the original space into a simpler or more special space (*e.g.*, from graphs to trees, from high-dimensional spaces to low-dimensional spaces). Then we only need to solve the same problem on this simple embedding space, because *metric embedding* techniques theoretically guarantee that the distance of any two points on the embedding space is closed to the distance of the corresponding points on the original space [7]. Formally speaking, *distortion* [7] is used to denote the maximum stretch of distances between the embedding space and the original space.

Hierarchically Separated Tree (HST), which was first introduced by Bartal in FOCS'96 [8], is one of the most prevalent choices for the embedding space. This is because the distortion of an HST is theoretically guaranteed. Specifically, Bartal proved that the lower bound of the guarantee was $\Omega(\log n)$ in [8] and the distortion guarantee was $O(\log n \log\log n)$ in [9], where n is the number of data points. This series of theoretical work culminated in the breakthrough of Fakcharoenphol, Rao and Talwar [10], who proposed the state-of-the-art algorithm to construct an HST with a tight guarantee $O(\log n)$.

In practice, many existing studies construct the HST as an index of the geometric data by the algorithm in [10] and design approximation algorithms for their applications via the HST. For instance, a few selected applications include clustering [11], [12], task assignment [13]–[15], trip planning [16], [17], privacy preservation [18], [19], information routing in wireless sensor networks [20], [21], distributed query processing [22], temporal data mining [23], facility location planning [24], [25].

Despite the popularity in the real-world problems, much less attention has been paid to find an efficient construction method. Specifically, the STOC'03 paper [10] and its long version [26] claimed that their construction algorithm could be implemented in $O(n^3)$ time and a more careful implementation could be made to run in $O(n^2)$ time, where $O(n^2)$ is the optimal time complexity in the *worst-case*. However, no details of the $O(n^2)$-time construction have ever been given and we have proved their implementation takes at least $O(n^3)$ time in this paper. In terms of the *average-case* time complexity, an $O(n^2 \log n)$-time algorithm has been given in [27]. Moreover, the space complexity was not discussed in these studies [10], [26], [27]. Thus, *it still remains elusive to construct an HST with optimal time and space complexity (in the worst-case).*

Moreover, existing studies usually assume the geometric data is static and predefined, and hence do not provide solutions for handling ***insertions*** (deletions can be trivially supported). The assumption may be impractical. For instance, in a wireless sensor network, an HST is constructed from the sensors' locations for information routing [21], where a new sensor may also be added in the network. Another example is that an HST is constructed by the workers' locations in spatial crowdsourcing [13], where a new worker can appear anywhere at anytime. Though insertions can be supported by reconstructing the HST, this method is inflexible in practice.

In this paper, we study the *Embedding Arbitrary metrics by Tree metrics* (EAT) problem. The EAT problem embeds an arbitrary metric space into a tree metric that minimizes the distortion. To improve the efficiency of *construction*, we apply dynamic programming to reduce the time complexity to $O(n^2)$ and propose a compressing strategy to reduce the space cost to $O(n)$. Since $O(n^2)$ is the optimal time complexity, we also design a pruning strategy to accelerate the running time. To flexibly support *insertion*, we propose a new data structure, called Hierarchically Separated Forest (HSF), which consists of multiple HSTs based on the disjoint point sets. Thus, when a few points are inserted, only one of the HSTs needs to be updated, and hence reconstructing the whole tree is avoided.

Our main contributions are summarized as follows.

- Our construction method significantly improves the time and space complexity (from $O(n^3)$ and $O(n^2)$ to $O(n^2)$ and $O(n)$ respectively), which is optimal.
- We propose Hierarchically Separated Forest to flexibly support insertions and prove that it has a tight theoretical guarantee ($O(\log n)$) in the distortion.
- Extensive experiments show that our construction method is up to 29.8× faster than the state-of-the-art method while saving up to 94.7% spaces, and our insertion method outperforms the baseline in terms of effectiveness and running time (*e.g.*, by up to 491× faster).

The rest of our paper is organized as follows. We first introduce the EAT problem in Sec. II and then present the state-of-the-art construction method in Sec. III. We next elaborate on our construction algorithms in Sec. IV and insertion algorithm in Sec. V. Finally, we conduct experiments in Sec. VI, review related studies in Sec. VII and conclude in Sec. VIII.

II. Problem Statement

In this section, we introduce the basic concepts in Sec. II-A and the problem definition in Sec. II-B.

A. Preliminaries

Definition 1 (Metric Space [7]): A metric space ("metric" for short) is denoted by $\mathcal{S} = (\mathcal{V}, \mathcal{D})$, where $\mathcal{V}$ is a set of points and $\mathcal{D} : \mathcal{V} \times \mathcal{V} \to [0, +\infty)$ is a distance function satisfying the following conditions for any points $x, y, z \in \mathcal{V}$:

(1) **Identity of indiscernibles**: $\mathcal{D}(x, y) = 0 \Leftrightarrow x = y$;
(2) **Symmetry**: $\mathcal{D}(x, y) = \mathcal{D}(y, x)$;
(3) **Triangle inequality**: $\mathcal{D}(x, y) + \mathcal{D}(y, z) \geq \mathcal{D}(x, z)$.

The metric spaces are ubiquitous in real-world applications, *e.g.*, Euclidean spaces and Manhattan distance. The points $\mathcal{V}$ of the metric space $\mathcal{S}$ can represent a set of data objects or spatial locations, and the distance function $\mathcal{D}(\cdot, \cdot)$ can be either Euclidean distance or Manhattan distance. We use n to denote the number of points in $\mathcal{V}$. Though $\mathcal{D}(\cdot, \cdot)$ is named as a distance function, it can be replaced with other function in practice, *e.g.*, the travel time between two locations or similarity between two objects.

Definition 2 (Tree Metric [8]): A tree-based metric ("tree metric" for short) $\mathcal{S}_T = (\mathcal{V}_T, \mathcal{D}_T)$ represents a tree-based metric space with positive edge weight, where the distance $\mathcal{D}_T(\cdot, \cdot)$ between any two points is the sum of the edge weights along their shortest path.

In this paper, we focus on the tree metric, since it is widely-used as the "simpler" metric space in metric embedding [4], [7]–[10], [20], [26]. The basic concepts in metric embedding techniques are formally defined as follows.

Definition 3 (Embedding, Stretch and Distortion [8]): Given two metric spaces $\mathcal{S} = (\mathcal{V}, \mathcal{D})$ and $\mathcal{S}_T = (\mathcal{V}_T, \mathcal{D}_T)$, a mapping $f : \mathcal{V} \to \mathcal{V}_T$ is called an **embedding** if the following condition is satisfied for any two points $x, y \in \mathcal{V}$:

$$\mathcal{D}(x, y) \leq \mathcal{D}_T(f(x), f(y)). \tag{1}$$

In this embedding, the **stretch** of the distance between any two points $x, y \in \mathcal{V}$ is defined as

$$\text{stretch}(x, y) = \frac{\mathcal{D}_T(f(x), f(y))}{\mathcal{D}(x, y)}. \tag{2}$$

Accordingly, the **distortion** of this embedding is the maximum of all the stretches, *i.e.*,

$$\text{distortion} = \max_{x \in V, y \in V - \{x\}} \frac{\mathcal{D}_T(f(x), f(y))}{\mathcal{D}(x, y)} \tag{3}$$

Here, we call the input metric space $\mathcal{S}$ as the *original metric*. According to Equation (3), the *distortion* represents the upper bound of the stretches and it takes at least $O(n^2)$ time (*i.e.*, enumerating the pairwise stretches) to calculate the distortion. In practice, a smaller distortion is usually more desired.

B. Problem Definition

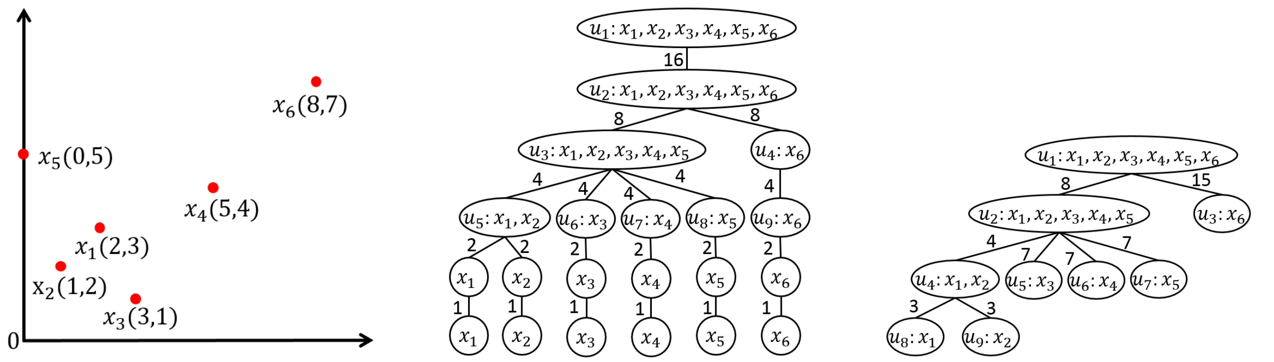
Based on the aforementioned, we present the *Embedding Arbitrary metrics by Tree metrics* (EAT) problem as follows.

Definition 4 (EAT Problem [8]): Given an original metric $\mathcal{S} = (\mathcal{V}, \mathcal{D})$, we aim to embed $\mathcal{S}$ into a tree metric $\mathcal{S}_T = (\mathcal{V}_T, \mathcal{D}_T)$ by a mapping $f : \mathcal{V} \to \mathcal{V}_T$ that minimizes the **distortion** of the embedding in Equation (3).

Then we illustrate the EAT problem through a toy example.

Example 1: In Fig. 1a, there are six points x_1-x_6 in an Euclidean space (*i.e.*, original metric). The EAT problem aims to construct a tree metric to minimize the distortion. Fig. 1b illustrates a feasible solution since the distance for any two points on the tree is no shorter than their Euclidean distance. For example, for points x_4 and x_6, the distance on the tree is $\mathcal{D}_T(x_4, x_6) = 1+2+4+8+8+4+2+1 = 30$ and the Euclidean distance is $\mathcal{D}(x_4, x_6) = \sqrt{(5-8)^2 + (4-7)^2} = 4.24$ (< 30). By Equation (2), we can calculate the stretch of the distance between x_4 and x_6 as $\frac{\mathcal{D}_T(x_4, x_6)}{\mathcal{D}(x_4, x_6)} = 7.07$. Similarly, we also enumerate the other stretches. By Equation (3), the distortion of the embedding is the maximum stretch among these values, *i.e.*, $\frac{\mathcal{D}_T(x_4, x_6)}{\mathcal{D}(x_4, x_6)} = 7.07$. Fig. 1c is also a feasible solution with the same distortion. Indeed, Fig. 1b and Fig. 1c illustrate the standard HST in [10] (see Sec. III) and the compact HST proposed by our paper in Sec. IV.

We next present the hardness results of the EAT problem in Theorem 1, which has been proved by pioneer work [8].



(a) An instance of the Euclidean space (b) The standard 2-HST [10] (c) The compact 2-HST (our paper)

Fig. 1: A toy example of the EAT problem, where the construction parameters are $\pi = \{x_1, \cdots, x_6\}$ and $\beta = 0.5$

Theorem 1: The EAT problem is NP-hard. The distortion of any deterministic algorithm is $\Omega(n)$ when the original metric space is the n-cycle (*i.e.*, Theorem 6 in [8]).

Due to the hardness results, existing studies [8]–[10], [26] (and our paper) focus on the approximation solutions instead of optimal solutions. For instance, the distortions of the HSTs as shown in Fig. 1 are not minimum. Moreover, they usually focus on designing randomized algorithms instead of the deterministic approaches. To analyze the theoretical guarantees of the randomized algorithms, pioneer work [8] first introduces probabilistic approximation, which is a standard evaluation in the studies on metric embedding [4], [6]–[10], [20], [26].

Definition 5 (Probabilistic Approximation [8]):] The tree metric $\mathcal{S}_T$ is said to be ρ-probabilistically approximates the original metric $\mathcal{S}$ if there is a probability distribution over $\mathcal{S}_T$ such that for any two points $x, y \in \mathcal{V}$, $\mathbb{E}[\mathcal{D}_T(f(x), f(y))] \leq \rho \times \mathcal{D}(x, y)$, where $\mathbb{E}[\cdot]$ denotes the expectation.

To obtain the tight guarantee of the expected distortion, *Hierarchically Separated Tree (HST)* is the most prevalent solution, which will be elaborated in the next section.

III. Hierarchically Separated Tree and Existing Construction Method

In this section, we present the definition of the HST in Sec. III-A and the existing construction method in Sec. III-B.

A. Hierarchically Separated Tree (HST)

HST was first defined by Bartal in [8] as follows.

Definition 6 (HST/k-HST): A hierarchically separated tree (HST) is a rooted weighted tree (with a user-defined parameter k and a height $\mathcal{H}$) that satisfies the following properties:

(1) For any integer $i = 1, \cdots, \mathcal{H}$, the edges between ith level and $(i+1)$th level have the same weight;
(2) For any integer $i = 2, \cdots, \mathcal{H}$, the edge weight between ith level and $(i+1)$th level equals to the edge weight between $(i-1)$th level and ith level divided by the parameter k;
(3) An HST has exactly n leaves and each of them represents a unique point of the original metric space.

According to the *first* property, any node u at the same level has the same edge weight to its children (denoted by $\mathsf{chi}(u)$). In the *second* property, a constant parameter k (≥ 2) affects the edge weight on the HST and different values of k can lead to different structures of the HST. Thus, the HST is also known as k-HST. In general, $k = 2$ (*i.e.*, 2-HST) is the most prevalent in existing studies. The *third* property of HST defines the distance function $\mathcal{D}_T(\cdot, \cdot)$ on the HST. Specifically, let $\mathsf{path}(u, v)$ denote the path between the leaf u and the leaf v on the HST and the distance from u to v can be calculated as $\mathcal{D}_T(u, v) = \sum_{e \in \mathsf{path}(u,v)} \mathcal{W}(e)$, where $\mathcal{W}(e)$ denotes the weight of edge e on the HST. The path from u to v consists of two parts, the path from u to their lowest common ancestor (denoted by $\mathsf{lca}(u, v)$) and the path from $\mathsf{lca}(u, v)$ to v.

The expected **distortion** ("distortion for short") of an k-HST is $O(k \log_k n)$ [10], [26]. Since k is often a constant in practice, we also use $O(\log n)$ to denote the distortion of an HST in the following. Though HST has a tight guarantee [10], [26], its construction still takes high complexity.

B. Existing Construction Method

In Algorithm 1, we review the construction method in [10], [26]. Algorithm 1 is not only the textbook algorithm (*e.g.*, see Chapter 8 in [5], Chapter 15 in [6], Chapter 18 in [4], and Chapter 6 in [21]), but still widely used as the construction procedure in recent studies (*e.g.*, [11], [16]–[19], [25]). Although Blelloch *et al.* [27] also propose a construction method (denoted by SeqFRT) with the guarantee of $O(\log n)$, we use Algorithm 1 as our baseline due to these reasons: (1) Our paper focuses on the worst-case complexity of the sequential algorithms and they both have the same time complexity in the worst-case. (2) Without parallelization, SeqFRT is always less efficient than Algorithm 1 in our datasets (please refer to Appendix D in our full paper [28] for the experimental results).

Basic Idea. The construction procedure is based on hierarchical partitions of the points in the original metric. Specifically, at each level, we use n circular ranges $\{\mathsf{B}(x, r) \mid \forall x \in \mathcal{V}\}$ to partition the original metric, where x denotes the center point and r denotes the radius. Each node of the HST represents a subset of points that are within the same circular ranges. We also randomly generate a permutation π of all the center points to determine the rank of the ranges. For example, the ith point in π (denoted by $\pi[i]$) has higher rank than the $(i+1)$th point in π (denoted by $\pi[i+1]$). Thus, when a point is in multiple circular ranges at the same time, we classify the point in the range with the highest rank of the center point. Moreover, the circular ranges at the same level have the same length in their

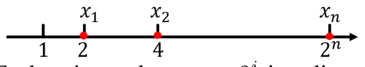
Algorithm 1: The state-of-the-art construction method [10], [26] (denoted by BASE)

```
input : the original metric $\mathcal{S} = (\mathcal{V}, \mathcal{D})$
output: a $k$-HST metric $T$
1  Choose a random permutation $\pi$ of the $n$ points in $\mathcal{V}$;
2  Choose $\beta \in [1/k, 1]$ uniformly at random;
3  Diameter of the metric $\mathcal{S}$ is $\Delta \leftarrow \max_{x,y\in\mathcal{V}} \mathcal{D}(x, y)$;
4  Height of the HST $T$ is $\mathcal{H} \leftarrow \lceil \log_k (\Delta + 1)\rceil + 1$;
5  Root of the HST is $rt \leftarrow \mathcal{V}$, radius $r_1 \leftarrow k^{\mathcal{H}} \cdot \beta$;
6  The set of nodes at the first level is $U_1 \leftarrow \{rt\}$ ;
7  for level $i \leftarrow 2$ to $\mathcal{H} + 1$ do
8      radius $r_i \leftarrow r_{i-1}/k$, the set of nodes $U_i \leftarrow \emptyset$;
9      for $j \leftarrow 1$ to $n$ do
10         foreach node $u \in U_{i-1}$ do
11             A node $v \leftarrow$ the set of unmarked points in
               $u$ within the circular range $\mathsf{B}(\pi[j], r_i)$;
12             if $v$ is a non-empty set then
13                 Add $v$ to the children of $u$ with weight
                   $k^{\mathcal{H}-i+2} \cdot \beta$ and mark the points in $v$;
14                 Add $v$ to the set of nodes $U_i$;
15 return $T$ is created by the nodes in $U_1, \cdots, U_{\mathcal{H}+1}$;
```

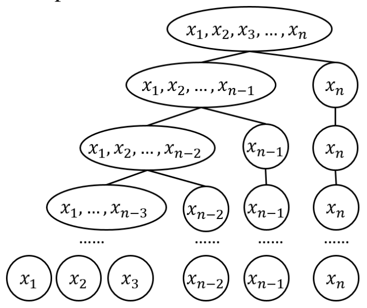
radii. The radius will decrease by a factor of k (*i.e.*, k-HST) at the next level. The partition will stop when the radius is below 1. As a common assumption is that the distance between any two points is no shorter than 1 (*e.g.*, by normalization), each leaf must correspond to a single point.

Algorithm Details. Algorithm 1 illustrates the detailed procedure. In lines 1-2, we randomly sample a permutation π of the n points and a parameter β. In lines 3-4, we calculate the diameter Δ and the height $\mathcal{H}$. In line 5, the root of the HST represents the whole set $\mathcal{V}$, *i.e.*, a partition with radius $r_1 = k^{\mathcal{H}} \cdot \beta > \Delta$. In line 6, we use U_i to denote the set of nodes on the HST at the ith level, and hence U_1 only contains the root of the tree. Lines 7-14 are the process of the hierarchical partition. At the ith level, the radius r_i decreases in line 8. We next iterate each center point $\pi[j]$ and partition the subspace consisting of the points in node $u \in U_{i-1}$ (lines 9-10). If some of the unmarked points in u are within the circular range $\mathsf{B}(\pi[j], r_i)$, we create a node v to represent these points in line 11. We also add node v to the children of u, mark these points, and update the node set U_i in lines 13-14. Overall, Algorithm 1 is a randomized algorithm due to the usage of the random parameters like π and β.

Example 2: Back to our example in Fig. 1. We assume $k = 2$, $\beta = 0.5$ and $\pi = \{x_1, \cdots, x_6\}$. In lines 3-6, we can calculate that $\Delta = \mathcal{D}(x_2, x_6) = 8.6$, $\mathcal{H} = 5$, $r_1 = 16$ and $U_1 = \{u_1\}$ (u_1 is the root). In lines 7-14, we construct the other nodes in Fig. 1b. For instance, when the level i is 3, the radius r_3 is 4 (line 8). In line 10, we iterate each node in the set $U_2 = \{u_2\}$. For example, when u_2 is iterated and $j = 1$, we check which points in u_2 are in the range $\mathsf{B}(\pi[1], r_3)$ (*i.e.*, a circular range centered at $\pi[1] = x_1$ with the radius $r_3 = 4$). In this example, only x_6 is not in this range (since $\mathcal{D}(x_6, x_1) = 7.2 > 4$). Thus, we create an internal node u_3 to represent the other points x_1-x_5 at the third level in Fig. 1b. Eventually, we can construct a 2-HST as shown in Fig. 1b.



(a) Each point x_i locates at 2^i in a line metric



(b) The constructed 2-HST of the line metric

Fig. 2: A worst-case of the complexity analysis

Complexity Analysis. We first present an instance to show Algorithm 1 takes at least $O(n^3)$ time and $O(n^2)$ space in the worst-case. Fig. 2a illustrates the original metric of the instance, where each point x_i locates at 2^i in a line. In this instance, when $\pi = \{x_1, x_2, \cdots, x_n\}$ and $\beta = 0.5$, the height of the 2-HST is $O(n)$ (see Fig. 2b). Specifically, at the ith level, the points $\{x_1, \cdots, x_{n-i+1}\}$ are within the same circular range centered at $\pi[1] = x_1$. The point x_{n-i+2} is within the circular range centered at $\pi[n - i + 1] = x_{n-i+1}$. The other points are within the circular range centered at itself. Thus, for any singleton node contained x_i, line 11 needs to be checked for at least i times from the $(n - i + 3)$th level and nth level (there are totally $i - 2$ levels). Overall, line 11 must have been checked for at least $\sum_{i=1}^{n} i(i - 2) = O(n^3)$ times. Moreover, as shown in Fig. 2b, the output HST has a height of n and there are k nodes in the kth level of the tree.

Rigorously speaking, when the distance function $\mathcal{D}$ takes $O(1)$ time and $\Delta \leq 2^{O(n)}$, the time complexity of Algorithm 1 is $O(n^3)$ and the space cost of the constructed HST is $O(n^2)$.

IV. Efficient Construction Method

Since existing method has a relatively high complexity, we propose a more efficient construction method in this section. which improves the time complexity to $O(n^2)$ and reduces the memory cost of the output tree to $O(n)$. Specifically, we first introduce our idea of *dynamic programming* to improve the time complexity (Sec. IV-A). Next, we elaborate our *compressing strategy* to reduce the memory cost in Sec. IV-B. Finally, we present the complete algorithm and discuss how to break the assumption ($\Delta \leq 2^{O(n)}$) in Sec. IV-C. The improved complexity is *optimal*, because any tree metric needs at least $O(n)$ space to store the n points and at least $O(n^2)$ time to calculate the distortion or scan the input (*e.g.*, when the input is given by a graph with an $O(n^2)$ distance matrix).

A. Dynamic Programming

Main Idea. Essentially, the baseline needs to test whether any point in $\mathcal{V}$ is within $O(n^2)$ circular ranges $\{\mathsf{B}(\pi[j], r_i) \mid 1 \leq j \leq n, 1 \leq i \leq \mathcal{H}+1\}$. Since the radius r_{i+1} is shorter than the

TABLE I: The values of $\mathsf{cen}[x][i]$ in Example 1

(a) $\mathsf{cen}[x][i]$ after step (1)

i	1	2	3	4	5	6
x_1	-	-	-	-	-	1
x_2	-	-	-	1	-	2
x_3	-	-	1	-	-	3
x_4	-	2	1	-	-	4
x_5	-	3	1	-	-	5
x_6	2	1	-	-	-	6

(b) $\mathsf{cen}[x][i]$ after step (2)

i	1	2	3	4	5	6
x_1	1	1	1	1	1	1
x_2	1	1	1	1	2	2
x_3	1	1	1	3	3	3
x_4	1	1	1	4	4	4
x_5	1	1	1	5	5	5
x_6	1	1	6	6	6	6

radius r_i in Algorithm 1, a point within $\mathsf{B}(\pi[j], r_{i+1})$ implies that it is also within the ranges $\mathsf{B}(\pi[j], r_i), \cdots, \mathsf{B}(\pi[j], r_1)$. Thus, our main idea is to (1) find the smallest circular range centered at $\pi[j]$ with the shortest radius $r_{\mathcal{L}_j}$ to cover each point $x \in \mathcal{V}$, and (2) apply dynamic programming (DP) to obtain the same partitions as in Algorithm 1.

Naive DP. At the ith level of Algorithm 1, we assume that a point x is partitioned by the circular range whose center point is ranked with $\mathsf{cen}[x][i]$ in π. Then we can pre-process the array $\mathsf{cen}[x][i]$ in $O(n^2)$ time based on two cases: the *leaf level* and the *other levels*.

At the *leaf level* ($i = \mathcal{H}+1$), $\mathsf{cen}[x][\mathcal{H}+1]$ equals to the rank of x in π since each leaf node represents a single point (*i.e.*, the correct circular range can be only centered at x).

At the *other levels* ($i \le \mathcal{H}$), we first enumerate each point $x \in \mathcal{V}$ and then process $\mathsf{cen}[x][1 \dots \mathcal{H}]$ by these two steps.

(1) **Initialization.** For each center point $\pi[1], ..., \pi[x\text{'s rank}]$, we find the smallest circular range centered at $\pi[j]$ that still covers x. Here, x's rank is the rank of the point x in the permutation π. In other words, we determine the shortest radius in $r_1, \cdots, r_{\mathcal{H}+1}$ such that it is longer than the distance between $\pi[j]$ and x, We can then calculate the level (denoted by $\mathcal{L}_j$) of this radius by Equation (4) (because $r_{\mathcal{L}_j} \ge \mathcal{D}(\pi[j], x)$).

$$\mathcal{L}_j = \mathcal{H} + 1 - \lceil \log_k \left(\mathcal{D}(\pi[j], x)/\beta \right) \rceil \tag{4}$$

Finally, we set $\mathsf{cen}[x][\mathcal{L}_j]$ as j if it is still undefined. Otherwise, we know point x has been covered by a circular range whose center point has a higher rank than $\pi[j]$ (*i.e.*, we have gotten the correct value).

(2) **Naive DP.** Since some element in $\mathsf{cen}[x][\cdot]$ may be still undefined, we update $\mathsf{cen}[x][i]$ by the following DP equation for $i = \mathcal{H}+1, \cdots, 1$ (*i.e.*, by a reverse order).

$$\mathsf{cen}[x][i] = \min\{\mathsf{cen}[x][i],\ \mathsf{cen}[x][i+1]\} \tag{5}$$

The reasons are as follows: (1) the radius of the $(i+1)$th level is shorter than the radius of the ith level (*i.e.*, at the ith level, the point x can be covered by the circular range $\mathsf{B}(\pi[\mathsf{cen}[x][i+1]], r_i)$) (2) x belongs to the circular range that has a center point with the higher rank in π (*i.e.*, we use a $\min$ operator in Equation (5)).

Example 3: Table Ia and Table Ib illustrate the array $\mathsf{cen}[\cdot][\cdot]$ in Example 1 after the steps of *initialization* and *Naive DP*, respectively. In Table Ia, when iterating x_3 from the point set $\mathcal{V}$, we need to iterate j from 1 to 3 since the rank of x_3 in π is 3. Since $\mathcal{D}(\pi[1], x_3) = \mathcal{D}(x_1, x_3) = 2.24$ and $\mathcal{D}(\pi[2], x_3) = 2.24$, $\mathcal{L}_1 = 5 + 1 - \lceil \log_2 2.24/0.5 \rceil = 3$ and $\mathcal{L}_2 = 3$ by Equation (4). Thus, when $j = 1$, we set $\mathsf{cen}[x_3][\mathcal{L}_1]$ as 1. When $j = 2$, we do not replace $\mathsf{cen}[x_3][\mathcal{L}_2]$ with 2 since $\mathsf{cen}[x_3][3]$ has been defined. After the step of initialization, we update $\mathsf{cen}[x_3][1 \dots 6]$ by Equation (5) and can easily obtain the fourth row of Table Ib.

Improved DP. At the ith level, we only use one column of this array, *i.e.*, $\mathsf{cen}[\cdot][i]$. Thus, if this column can be dynamically obtained from the previous column (*e.g.*, $\mathsf{cen}[\cdot][i-1]$), we will only need $O(n)$ space (instead of $O(n^2)$ space) to store cen. Here, we utilize the fact that the radii of the circular ranges are monotonically decreasing with the increase of level. Specifically, at the ith level, if a point x is still in the smaller circular range centered at point $\pi[\mathsf{cen}[x][i-1]]$, then we have $\mathsf{cen}[x][i] = \mathsf{cen}[x][i-1]$. Otherwise, $\mathsf{cen}[x][i]$ needs to be updated, *e.g.*, we keep enumerating the center points from $\pi[\mathsf{cen}[x][i-1]+1]$ to $\pi[n]$ until finding the first center point whose circular range with radius r_i can cover x. In summary, $\mathsf{cen}[x][i]$ is updated from $\mathsf{cen}[x][i-1]$ by Equation (6).

$$\mathsf{cen}[x][i] = \tag{6}$$
$$\begin{cases} \mathsf{cen}[x][i-1], & \text{if } \mathcal{D}(x, \pi[\mathsf{cen}[x][i-1]]) \le r_i \\ \text{minimum } j > \mathsf{cen}[x][i-1] \text{ s.t. } \mathcal{D}(x, \pi[j]) \le r_i, & \text{otherwise} \end{cases}$$

Example 4: By the *naive DP*, we obtain Table Ib row by row in Example 3. By the *improved DP*, we illustrate how to obtain Table Ib column by column. Specifically, the column $\mathsf{cen}[x][1]$ is initialized with 1, since $\mathcal{D}(x, \pi[1]) \le r_1$ for each point x. When level $i = 2$, $\mathsf{cen}[x][2]$ is still 1 since the distance between $\pi[1] = x_1$ to any point x is no longer than the radius $r_2 = 8$. When $i = 3$, only x_6 is not in the circular range of $\pi[1]$ with radius $r_3 = 4$ anymore. Thus, we update $\mathsf{cen}[x_6][3]$ by Equation (6). Specifically, we enumerate j from 2 to 6 and pick the minimum integer j (=6) such that $\mathcal{D}(x_6, \pi[j]) < r_3$. In other words, x_6 belongs to the circular range of $\pi[6]$ at the third level. For other points, $\mathsf{cen}[x][3]$ is still 1. Similarly, we can obtain the same array as shown in Table Ib.

Since we only need the ith column to update the $(i+1)$th column, we only need an $O(n)$ array (denoted by $\mathsf{cen}[x]$) to maintain the right circular range for each point.

B. Compressing Strategy

Preliminary. An HST takes $O(n^2)$ space, since it has many redundant nodes (*e.g.*, at least 85% of the nodes are redundant in our experiments), which are defined as follows.

Definition 7 (Redundant Node): An internal node of the HST is redundant if *(1)* the node is not the root node, and *(2)* the node has only one child node.

The redundant nodes are produced since the *standard* HST (see Definition 6) restricts the edge weight at each level. However, the EAT problem itself has no restriction on the tree metric, and hence these nodes are unnecessary for obtaining the theoretical results. In the following, we define the *compact* HST based on the definition of the *standard* HST.

Definition 8 (Compact HST): A compact HST is a rooted weighted tree that satisfies the following properties:

(1) The compact HST has no redundant nodes;

Algorithm 2: Our construction method HST+DPO

```
1  Randomly sample a permutation $\pi$ and $\beta \in [1/k, 1]$;
2  $\Delta \leftarrow \max_{y\in\mathcal{V}} \mathcal{D}(\pi[1], y)$, $\mathcal{H} \leftarrow \lceil \log_k(\Delta+1)\rceil + 1$;
3  Root $rt \leftarrow \mathcal{V}$, radius $r_1 \leftarrow k^{\mathcal{H}} \cdot \beta$, node set $U_1 \leftarrow \{rt\}$;
4  $\mathsf{cen}[1\ldots n] \leftarrow \{1, 1, \cdots, 1\}$; // Initialization
5  for level $i \leftarrow 2$ to $\mathcal{H}+1$ do
6      Radius $r_i \leftarrow r_{i-1}/k$, the set of nodes $U_i \leftarrow \emptyset$;
7      foreach point $x$ in $\mathcal{V}$ do    // Improved DP
8          Update $\mathsf{cen}[x]$ by Equation (6);
9      foreach node $u$ in $U_{i-1}$ do
10         $\mathsf{vec} \leftarrow \mathsf{vec} \cup \{(\mathsf{cen}[x], x, u) \mid$ point $x \in u\}$;
11     Counting-sort $\mathsf{vec}$ by the $\mathsf{cen}$ in each triple;
12     foreach continuous interval in $\mathsf{vec}$ such that each
        point $x$ was contained in the same node $u$ do
13         A node $v \leftarrow$ the points of this interval;
14         Add $v$ to the children of $u$ with weight
            $k^{\mathcal{H}-i+2} \cdot \beta$ and add $v$ to the set of nodes $U_i$;
15     foreach redundant $u \in U_{i-1}$ do    // Compress
16         $w_0 \leftarrow$ edge weight between $u$ and $\mathsf{par}(u)$;
17         In the children of $\mathsf{par}(u)$, replace $u$ with $u$'s
            child node with edge weight $w_0 + k^{\mathcal{H}-i+2} \cdot \beta$;
18 return $T$ is created by the nodes in $U_1, \cdots, U_{\mathcal{H}+2}$;
```

(2) For any integer $i = 2, \cdots, \mathcal{H}$, the edge weight between ith level and $(i+1)$th level is smaller than the edge weight between $(i-1)$th level and ith level;
(3) The compact HST has exactly n leaves and each of them represents a different point in the original metric space.

In a compact HST, the degree of any internal node (except the root) is at least 2 and the number of leaves is n. Thus, the space cost of a compact HST is $O(n)$.

Main Idea. To construct a compact HST, we can first construct a standard HST and then compress it. However, an $O(n^2)$ space is still needed to store the standard HST before compression. To break the memory bottleneck, our main idea is to *keep merging* the redundant nodes during the construction procedure. For instance, if we want to merge node u and its child node v, we can replace u to v in the parent node of u (denoted by $\mathsf{par}(u)$), and then update the edge weight of $(v, \mathsf{par}(u))$ by the sum of following two parts: the edge weight *between* u *and* $\mathsf{par}(u)$ and the edge weight *between* v *and* u.

Example 5: As shown in Fig. 1b, the nodes u_1 and u_2 are not redundant based on Definition 7. However, after the nodes at the 4th level are created, u_4 becomes redundant since it has only one child node u_9. To remove the redundancy, we replace u_4 with u_9 as a child of u_2 and the edge weight becomes $8+4=12$, where 8 is the edge weight between u_4 and $\mathsf{par}(u_4) = u_2$ and 4 is the edge weight between u_9 and u_4.

C. Algorithm Details

Details. Algorithm 2 illustrates our construction algorithm. Lines 1-3 are similar to the lines 1-6 of Algorithm 1. The only difference is that we set Δ as $\max_{y\in\mathcal{V}} \mathcal{D}(\pi[1], y)$ instead of $\max_{x,y\in\mathcal{V}} \mathcal{D}(x, y)$ because of our pruning strategy in Lemma 1. Next, we initialize $\mathsf{cen}[\cdot]$ with 1 in line 4 and update $\mathsf{cen}[x]$ by Equation (6) in lines 7-8. In lines 9-14, we construct the ith level of the HST based on $\mathsf{cen}[\cdot]$. Specifically, in lines 9-10, we create a vector of triples (denoted by vec), where each triple contains point x, the node u (contained x) at the $(i-1)$th level, and $\mathsf{cen}[x]$. In line 11, we sort vec by the counting-short based on the $\mathsf{cen}[x]$ (between 1 and n) in each triple. After that, the points, which are partitioned by the same circular range, will appear closely in the sorted vector. Moreover, since counting-sort is a *stable sorting* algorithm, we also separate these triples into multiple intervals such that the points (x) of these triples are contained in the same internal node (u) at the $(i-1)$th level. In lines 12-14, we create a node v for each of these intervals and add v to the children of u with the same edge weight as the baseline. In lines 15-17, we remove the redundant nodes in U_{i-1} based on our compressing strategy. Eventually, we construct a compact HST.

Example 6: Back to Example 1. In line 2, We can calculate that $\Delta = \max_{y\in\mathcal{V}} \mathcal{D}(\pi[1], y) = 7.2$. Thus, the height $\mathcal{H}$ is $\lceil \log_2 7.2\rceil + 1 = 4$, which is lower than the height (5) by Algorithm 1. The values of $\mathsf{cen}[\cdot]$ at levels 2-5 correspond to the columns 3-6 ($i = 3, ..., 6$) of Table Ib. Accordingly, we can construct the nodes of the ith level in Fig. 1c. For example, when $i = 3$, we create a vector $\mathsf{vec} = \{(1, x_1, u_2), (1, x_2, u_2), (3, x_3, u_2), (4, x_4, u_2), (5, x_5, u_2), (6, x_6, u_3)\}$ in lines 9-10. After the counting-sort in line 11, we obtain the following five intervals: $\{(1, x_1, u_2), (1, x_2, u_2)\}$, $\{(3, x_3, u_2)\}$, $\{(4, x_4, u_2)\}$, $\{(5, x_5, u_2)\}$, and $\{(6, x_6, u_3)\}$. As shown in Fig. 1c, the first four intervals correspond to the internal nodes u_4-u_7, which are the children of u_2. For the last interval, we create a (virtual) node u_3', which is the only child of u_3. Then u_3 becomes redundant and we will compress u_3 and u_3' in lines 16-17. Eventually, we construct the compact HST in Fig. 1c.

Pruning. To further speed up the running time, we also apply the following pruning strategies in Algorithm 2.

Lemma 1: Our pruning strategies include

(1) In line 2, we can safely set Δ with $\max_{y\in\mathcal{V}} \mathcal{D}(\pi[1], y)$ instead of $\max_{x,y\in\mathcal{V}} \mathcal{D}(x, y)$.
(2) In line 8, we can safely skip any integer j such that $\mathsf{cen}[\pi[j]]$ is larger than the rank of x in π.

Proof: For the *first* strategy, when the center point is $\pi[1]$, the radius r_1 with our pruning strategy is long enough to cover all the points at the first level. Thus, the tree constructed with the pruning strategy (denoted by T^*) is indeed a subtree of the HST T constructed without pruning. Moreover, any path between the leaves will not traverse the nodes between the root of T and the root of the subtree T^*. Thus, T^* has the same distance function as T.

For the *second* strategy, the radius r_i decreases with the increase of level in the construction methods. As $\mathsf{cen}[\pi[j]]$ is larger than the rank of x, it means the distance between x and $\pi[j]$ must be larger than r_i. Hence we safely skip j. ■

Correctness and Approximation. We next prove the correctness and theoretical guarantee of Algorithm 2 by Theorem 2.

Theorem 2: Algorithm 2 always outputs a feasible tree metric and the distortion guarantee of Algorithm 2 is $O(\log n)$.

Proof: We first prove our *DP strategy* does not change the distance function by proving the value of $\mathsf{cen}[\cdot]$ at each level is exactly the same as in Algorithm 1. In line 8, Algorithm 2 tries each center point from $\mathsf{cen}[x]$ to the rank of x in π. Thus, if Algorithm 1 uses any center point in this range, Algorithm 2 can also find the correct one. Then, we only need to prove that the center point in Algorithm 1 cannot be any point before $\mathsf{cen}[x]$ in the permutation π. It is true for the first level since the circular range centered at $\pi(1)$ can cover all the points at this level. For the other levels, we suppose to the contrary, *i.e.*, there exists one such $\pi(j) < \mathsf{cen}[x]$ such that the $\mathcal{D}(x, \pi(j))$ is shorter than the radius r_i. Thus, the value of $\mathsf{cen}[x]$ at the $(i-1)$th level should be no larger than j since $\mathcal{D}(x, \pi(j))$ should also be shorter than the radius r_{i-1} $(> r_i)$. By Equation (6), the value of $\mathsf{cen}[x]$ at the ith level cannot be larger than j, which is contradicted to the assumption.

We next prove our *compressing strategy* does not change the distance function. The compact HST is converted from a standard HST by merging the redundant nodes. When merging the redundant nodes, we also adding the weight of the removed edge to the remained edge. As a result, the total sum of the edge weight is not changed. Since the distance function of a tree metric is defined as the total sum of the edge weight (Definition 2), the distance function also remains the same.

Since all these strategies (including Lemma 1) will not change the distance function, Algorithm 2 always outputs a feasible tree metric (*i.e.*, the compact HST) and its distortion guarantee is same as the distortion guarantee of the standard HST (*i.e.*, $O(\log n)$ [10]) based on Definition 5. ■

Complexity Analysis. Specifically, when $\Delta \leq 2^{O(n)}$, there are $O(n)$ levels in line 5. In each iteration, the algorithm may update cen in lines 7-8. Since the value of $\mathsf{cen}[x]$ is between 1 and n for each point x, the array cen is updated at most $O(n^2)$ times in all these levels. Thus, lines 5-6 take $O(n^2)$ time in all the $O(n)$ levels. Besides, in each level, iterations 9-10, line 11, iterations 12-14 and iterations 15-17 take $O(n)$ time. Thus, the *time complexity* is improved from $O(n^3)$ to $O(n^2)$. Moreover, the *space consumption* of the output tree reduces from $O(n^2)$ to $O(n)$, because the size of a compact HST is $O(n)$ and $\mathsf{cen}[\cdot]$ also takes $O(n)$ space.

Extension. Finally, we discuss how to extend Algorithm 2 when the value of Δ is unbounded in Theorem 3.

Theorem 3: When Δ is unbounded, we can extend Algorithm 2 as follows to keep the time and space complexity:

(1) Let $I = \{i | k^{\mathcal{H}-i}\beta \leq \mathcal{D}(x, y) \leq k^{\mathcal{H}-i+1}\beta, \forall x, y \in \mathcal{V}\}$;

(2) We iterate each level i from I in a descending order;

(3) For each level, the edge weight $k^{\mathcal{H}-j+2}\beta$ is changed into $\sum_{j=i'+1}^{i}(k^{\mathcal{H}-j+2}\beta)$, where i' denotes the previously iterated level and the initial value of i' is 1.

Proof: We first prove that the extension does not change the distance function of a compact HST. Since the extension only iterates the levels in I instead of all the values between 1 and $\mathcal{H}+1$, we only need to prove that we can safely skip the levels outside I. WLOG, we assume any level $\mathcal{L} \in (i', i)$ and prove $\mathcal{L}$ can be safely skipped by our extension. Specifically, we know $\mathcal{D}(x, \pi[\mathsf{cen}[x]]) \leq r_{i'}$ at the i'th level, where $r_{i'} = k^{\mathcal{H}-i'+1}\beta$. Since $\mathcal{L}$ is not in the set I, $\mathcal{D}(x, \pi[\mathsf{cen}[x]])$ will be smaller than $r_{\mathcal{L}}$, where $r_{\mathcal{L}} = k^{\mathcal{H}-\mathcal{L}+1}\beta$. Otherwise, there will be at least one integer between i' and i in I, which is contradicted to the definition of i'. Accordingly, every point will be partitioned by the same center point $\mathsf{cen}[\cdot]$ between level $i'+1$ and level $i-1$. Therefore, the node set $U_{\mathcal{L}}$ will be the same as previous node set $U_{\mathcal{L}-1}$. Based on the definition of *redundant nodes*, these nodes are also redundant. Therefore, we use $\sum_{j=i'+1}^{i-1}(k^{\mathcal{H}-j+2}\beta)$ to denote the accumulated edge weight in line 17. After adding the edge weight of the ith level $(k^{\mathcal{H}-i+2}\beta)$, the distance function will remain the same.

We next prove the time complexity is still $O(n^2)$ and the space cost is $O(n)$. Since the size of I is $O(n)$ (which will be proved in Lemma 2), it takes $O(n^2)$ time and $O(n)$ space to calculate I (*e.g.*, by hash). Besides, we still iterate $O(n)$ levels and $\sum_{j=i'+1}^{i}(k^{\mathcal{H}-j+2}\beta)$ can be calculated in $O(1)$ time (by the sum formula of a geometric series). Therefore, the time complexity is still $O(n^2)$ and the space cost is still $O(n)$. ■

Lemma 2: The size of the set I in Theorem 3 is $O(n)$.

Proof: Our proof is based on the fact from [4] in Section 26.6.2(b) of Chapter 26 (Finite Metric Spaces and Partitions).

Fact 1: Let $\mathcal{S} = (\mathcal{V}, \mathcal{D})$ be a n-point metric space, and consider the set $J = \{j | 2^j \leq \mathcal{D}(x, y) \leq 2^{j+1}, \text{ for } x, y \in \mathcal{V}\}$. We have $|J| = O(n)$ [4].

First, we rewrite the definition of I: $I = \{i | \mathcal{H} + \log_k \beta - \log_k \mathcal{D}(x, y) \leq i \leq \mathcal{H} + 1 + \log_k \beta - \log_k \mathcal{D}(x, y), \forall x, y \in \mathcal{V}\}$. Accordingly, the difference between i's lower bound $\mathcal{H} + \log_k \beta - \log_k \mathcal{D}(x, y)$ and i's upper bound $\mathcal{H} + 1 + \log_k \beta - \log_k \mathcal{D}(x, y)$ is 1. If we define an integer set $P = \{\lceil \mathcal{H} + \log_k \beta - \log_k \mathcal{D}(x, y) \rceil \mid \forall x, y \in \mathcal{V}\}$, we can derive that $|I| \leq 2|P|$ (since $I \subseteq P \cup \{p + 1 \mid p \in P\}$).

Next, we prove $|P| = O(n)$. We rewrite the definition of J: $J = \{j | j \log_k 2 \leq \log_k \mathcal{D}(x, y) \leq (j+1) \log_k 2, \forall x, y \in \mathcal{V}\}$. Since the difference between $\log_k \mathcal{D}(x, y)$'s lower bound $j \log_k 2$ and its upper bound $(j+1) \log_k 2$ is $\log_k 2 \leq 1$ (the parameter $k \geq 2$), we can infer that $|P| \leq 2|J|$ (since $P \subseteq \{\lceil \mathcal{H} + \log_k \beta - j \log_k 2 \rceil \mid j \in J\} \cup \{\lceil \mathcal{H} + \log_k \beta - (j+1) \log_k 2 \rceil \mid j \in J\}$).

Finally, since $|J| = O(n)$, we can prove $|I| = O(n)$. ■

V. Efficient Insertion Method

In this section, we introduce a data structure called hierarchically separated forest (HSF) in Sec. V-A. Then we present our insertion algorithm by HSF in Sec. V-B. Deletion can be easily supported (see Appendix A in our full paper [28]).

A. Hierarchically Separated Forest

To support insertions, our baseline is to reconstruct an HST after the new points are inserted. We have this assumption because the partitions ($O(n)$ circular ranges) at each level ($O(n)$ level) can be changed at lot. For example, after deleting the original points and inserting the new points, the permutation π might be changed a lot. Since the construction procedure highly depends on the permutation π, each point can be partitioned by a different circular range at each level. To keep the $O(\log n)$ guarantee, we assume the HST is reconstructed. Thus, this baseline takes $O(n^2)$ time for each insertion.

To improve efficiency, our **main idea** is to maintain a forest, which consists of multiple HSTs constructed by disjoint point sets. As existing studies usually assume that there are a large number of predefined points in the original metric, we always hold still the HST (*e.g.*, T_1) constructed by these predefined points and only update the other HSTs (*e.g.*, $T_2, \cdots, T_c$) constructed by the newly inserted points. Thus, when querying the distance between any two points on HST T_i, the result can be directly calculated by the function D_{T_i}. Differently, when querying the distance between two points on the different HSTs (*e.g.*, T_i and T_j), we can project the point on T_j into one point (*e.g.*, its nearest neighbor) on T_i, and then calculate the distance. Specifically, we first define the hierarchically separated forest (HSF) as follows.

Definition 9 (HSF): A hierarchically separated forest (k-HSF) $\mathcal{F}$ is a set of c disjoint k-HSTs ($T_1, \cdots, T_c$) such that

(1) The HST T_1 is constructed by the predefined points $\mathcal{V}_1$;
(2) The other k-HSTs $T_2, \cdots, T_c$ are constructed by the newly inserted points (*e.g.*, $\mathcal{V}_2, \cdots, \mathcal{V}_c$), which may be deleted.
(3) The HSF also maintains the nearest neighbors in the point sets $\mathcal{V}_1, \cdots, \mathcal{V}_{i-1}$ for each point $x \in \mathcal{V}_i$, where $i = 2, \cdots, c$. The nearest neighbor is denoted by $\mathsf{NN}[x][i]$ and the corresponding distance is denoted by $\mathsf{NND}[x][i]$.
(4) The distance function of the forest is defined as follows:

$$\forall x, y \in V,\ \mathcal{D}_F(x, y) = \begin{cases} \mathcal{D}_{T_i}(x, y) & \text{if both } x, y \in \mathcal{V}_i \\ \mathcal{D}_{T_i}(x, \mathsf{NN}[y][i]) + \mathsf{NND}[y][i] & \text{if } x \in \mathcal{V}_i, y \in \mathcal{V}_j \ (i < j) \end{cases} \tag{7}$$

Approximation Analysis. By the data structure of k-HSF, we can still obtain a tight guarantee in the distortion ($O(\log n)$).

Theorem 4: The distortion guarantee of the k-HSF is still $O(k \log_k n) = O(\log n)$ by Equation (7).

Proof: Since n is the total number of points, the size of any set $\mathcal{V}_i$ is smaller than n. Let x, y be any two points and we prove the theorem from the following two cases.

(1) If both x and y are located in the same k-HST T_i, the distortion guarantee is $O(k \log_k |\mathcal{V}_i|) \leq O(k \log_k n)$.

(2) If points x and y are located in different k-HSTs (*e.g.*, T_i and T_j where $i < j$), the distance function becomes the second case in Equation (7). As $\mathsf{NN}[y][i]$ is the nearest neighbor of y in the point set $\mathcal{V}_i$, we know that $\mathcal{D}(y, \mathsf{NN}[y][i]) \leq \mathcal{D}(x, y)$, *i.e.*, $\mathsf{NND}[y][i] \leq \mathcal{D}(x, y)$. Thus, the distortion guarantee defined in Definition 5 is derived as follows.

$$\begin{aligned} &\frac{\mathbb{E}\left[\mathcal{D}_{T_i}(x, \mathsf{NN}[y][i]) + \mathsf{NND}[y][i]\right]}{\mathcal{D}(x, y)} \\ \leq\ & \frac{O(k \log_k |\mathcal{V}_i|) \cdot \left(\mathcal{D}(x, y) + \mathcal{D}(y, \mathsf{NN}[y][i])\right) + \mathcal{D}(x, y)}{\mathcal{D}(x, y)} \\ \leq\ & O(k \log_k |\mathcal{V}_i|) + \frac{O(k \log_k |\mathcal{V}_i|) \cdot \mathcal{D}(y, \mathsf{NN}[y][i])}{\mathcal{D}(x, y)} + o(1) \\ \leq\ & O(k \log_k |\mathcal{V}_i|) + \frac{O(k \log_k |\mathcal{V}_i|) \cdot \mathcal{D}(x, y)}{\mathcal{D}(x, y)} + o(1) \leq O(k \log_k n) \end{aligned}$$

■

Algorithm 3: Our insertion method HST+HSF

```
input : a $k$-HSF $\mathcal{F}$ and the inserting points $\mathcal{V}^+$
1 $i^* \leftarrow \arg\max_{i=2,\cdots,m} |\mathcal{V}_i|$, $\mathcal{V}_{i^*} \leftarrow \mathcal{V}_{i^*} \cup \mathcal{V}^+$;
2 $T_{i^*} \leftarrow$ re-construct the HST based on $\mathcal{V}_{i^*}$;
3 Update $\mathsf{NN}[x][\cdot]$, $\mathsf{NND}[x][\cdot]$ for any point $x \in \mathcal{V}_{i^*}$;
4 foreach point $x \in \mathcal{V}_j$ where $j > i^*$ do
5     Update $\mathsf{NN}[x][i]$ and $\mathsf{NND}[x][i]$ if the nearest
      neighbor of $x$ is from the inserted points $\mathcal{V}^+$;
```

Space Complexity. The space consumption of the HSF includes the space cost of these HSTs and the memory usage of the stored nearest neighbors. Let m denote the maximum number of points in the HSTs $T_2, \cdots, T_c$. Therefore, the space complexity of the HSF is $O(n + mc^2)$. Specifically, the space cost of the (compact) HSTs is $O(n)$ and the memory usage of the nearest neighbors is $O(m + \cdots + (c-1)m) = O(mc^2)$. In practice, we can set a small constant value for parameter c (*e.g.*, < 10) and hence the space complexity becomes to $O(n + m) = O(n)$.

B. Efficient Insertion Method

Basic Idea. When inserting some new points, our basic idea is to select the HST from $T_2, \cdots, T_c$ with the least points and re-construct it based on the new points and its original points. We also maintain the nearest neighbors of the other points.

Algorithm Details. Algorithm 3 illustrates the detailed procedure. In lines 1-2, we select the HST T_{i^*} with the least points and insert the new points $\mathcal{V}^+$ into this HST by re-constructing the HST T_{i^*}. In lines 3-5, we maintain the nearest neighbors NN and the corresponding distance NND for the HSTs $T_{i^*}, \cdots, T_c$.

Example 7: Back to Example 1. We assume the six points x_1-x_6 are predefined (*i.e.*, $\mathcal{V}_1 = \{x_1, \cdots, x_6\}$) and hence the (compact) HST T_1 is illustrated in Fig. 1c. We also assume an HSF consists of two HSTs, where T_2 is currently empty. To insert a point $x_7 = (3, 6)$, we first add it into point set $\mathcal{V}_2$ (line 1) and construct the HST T_2 based on $\mathcal{V}_2$. In line 3, we find the nearest neighbor of x_7 in the point set $\mathcal{V}_1$, *i.e.*, $\mathsf{NN}[x_7][1] = x_4$, $\mathsf{NND}[x_7][1] = \mathcal{D}(x_4, x_7) = 2.82$. When querying the distance between x_1 and x_7 on HSF, the result is $\mathcal{D}_{T_1}(x_1, \mathsf{NN}[x_7][1]) + \mathsf{NND}[x_7][i] = \mathcal{D}_{T_1}(x_1, x_4) + 2.82 = 16.82$ by Equation (7).

Time Complexity. Lines 1-2 take $O(m^2)$ to construct the HST by our DP-based method. Lines 3-5 take $O(cm \cdot |\mathcal{V}^+|) = O(m|\mathcal{V}^+|)$ time to update the nearest neighbors. Thus, the time complexity of Algorithm 3 is $O(m^2 + m|\mathcal{V}^+|)$.

VI. Experiments

In this section, we conduct experiments on construction in Sec. VI-A and experiments on insertion in Sec. VI-B. Finally, we summarize the major experimental findings in Sec. VI-C.

A. Experiments on Construction

1) Experimental Setup: **Datasets.** As many existing studies construct the HSTs from a 2D Euclidean space [13], [14], [16], [19], [20], we select four **real datasets** on the 2D Euclidean

TABLE II: Real datasets in 2-dimensional Euclidean space

Source	Foursquare		Didi Chuxing	
Dataset	*NYC*	*Tokyo*	*Chengdu*	*Haikou*
n	42,981	67,123	227,447	319,419
k	**2**,3,4,5,6			

TABLE III: Synthetic datasets under different distributions

Parameters	Values
$\mathcal{S}$	**10**-dimensional Euclidean space
Range	$[0, 10^7]$ for each dimension
n and k	**100k** and **2**
Uniform	$mean : [0.1, 0.3, \mathbf{0.5}, 0.7, 0.9]\ (\times 10^7)$
Normal	$\mu : [0.1, 0.3, \mathbf{0.5}, 0.7, 0.9]\ (\times 10^7)$, $\sigma : 0.2 \times 10^7$
Exponential	$1/\lambda : [0.1, 0.3, \mathbf{0.5}, 0.7, 0.9]\ (\times 10^7)$

space in Table II. The first two datasets, *NYC* and *Tokyo* [29], contain all the check-in locations in New York and Tokyo respectively, which are collected in Foursquare [30]. The other two datasets, *Chengdu* and *Haikou* [31], contain the origins and destinations of the car-hailing orders in Chengdu and Haikou, which are collected by Didi Chuxing [32]. These real datasets are widely used in existing studies [33]–[38].

We generate **synthetic datasets** to test the performance on *multi-dimensional data* under *different distributions*. As shown in Table III, we generate $100k$ points in 10-dimension Euclidean space and set the range of each dimension as $[0, 10^7]$ based on the real datasets. Similar to [39], in each dimension, we generate the values of these points under the Uniform, Normal and Exponential distributions (denoted by *Uni*, *Nor* and *Exp* respectively). We do not vary the number of dimensions, since the number of dimensions mainly affects the time cost to calculate the Euclidean distance, which is the same across all the compared algorithms.

We conduct **scalability tests** when n is varied in $[10^5, 3.2 \times 10^6]$ and k is 2. The n points are randomly generated in a 2D Euclidean space, where the range of each coordinate is $[0, 10^7]$.

Compared Algorithms. We compare our method HST+DPO (Algorithm 2) with the state-of-the-art method BASE (Algorithm 1). We also implement the naive version of Algorithm 2, called HST+DP, which does not use the compressing strategy and pruning strategy in Sec. IV. It only uses the naive DP (instead of improved DP) strategy in Sec. IV-A.

Metrics. All the algorithms are evaluated in terms of *running time* and *memory usage*. Since their output HSTs have the same distortions in each test case, we omit the results of their distortions due to the space limitation.

Implementation. All algorithms are implemented in C++ and the experiments are conducted on a server with 24 Intel 2.30GHz processors and 128GB memory. The average results of 30 times repeated experiments are reported.

2) Experimental Results: **Results on real datasets.** The first row of Fig. 3 shows the experimental results on *NYC* and *Tokyo*. In Fig. 3a and Fig. 3c, we observe that HST+DPO is the most efficient. For instance, HST+DPO is up to $24.7\times$ faster than BASE. It is also notably faster than HST+DP due to our pruning strategy. As for memory cost, HST+DPO is much more efficient than BASE and HST+DP is slightly less efficient than the baseline. The result is reasonable since HST+DP consumes an extra $O(n^2)$ space than the baseline and HST+DPO applies our compressing strategy. Specifically, HST+DPO has 78.3%-89.6% and 80.8%-90.7% lower memory cost than BASE and HST+DP, respectively. Besides, we also observe that the time cost and memory usage of BASE notably decrease with the increase of k while the results of HST+DPO are relatively stable. This is because (1) the height of the HST decreases with the increase of k and (2) BASE is more sensitive to the height than HST+DPO. In both datasets, the heights of the HSTs (*i.e.*, the number of iterations in line 7 in Algorithm 1) are between 11 and 27, which can be $O(n)$ in the worst-case. This explains the reason that the improvement of the running time by HST+DPO is not as large as n.

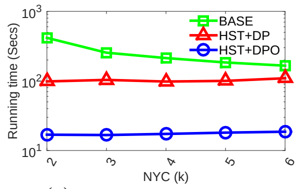
The second row of Fig. 3 shows the experimental results on *Chengdu* and *Haikou*. In terms of running time, HST+DPO is the most efficient and HST+DP is the runner-up, as shown in Fig. 3e and Fig. 3g. HST+DP and HST+DPO outperform the baseline BASE by up to $4.4\times$ and $16.6\times$ lower time cost, respectively. As for memory usage, HST+DPO is the most efficient, which needs up to 82.4% and 88.1% less memory spaces than BASE in *Chengdu* and *Haikou*, respectively. This is because 89% of the nodes in the standard HST are redundant. HST+DP takes a bit more spaces (<40MB) than BASE due to the usage of the array $\mathsf{cen}[\cdot][\cdot]$.

Results on synthetic datasets. Fig. 4a-Fig. 4f show the results on the synthetic datasets of Uniform, Exponential and Normal distributions. Overall, HST+DPO is always the most efficient in terms of both running time and memory usage. HST+DP is faster than BASE while it needs more spaces than BASE. Specifically, under the *Uniform* distribution, HST+DP and HST+DPO are $8.6\times$-$9.3\times$ and $26.3\times$-$28.8\times$ faster than BASE, respectively. Moreover, at least 93.8% of the space cost of the baseline can be reduced by HST+DPO. Under the *Exponential* distribution, HST+DPO is up to $29.8\times$ faster than BASE and it also reduces 94.7% of the memory cost of the baseline. Under the *Normal* distribution, HST+DP and HST+DPO need at least $8\times$ and $25\times$ shorter time than BASE. As for memory usage, HST+DPO needs less than 6% of the space cost of BASE, because it safely removes 95% of the nodes (*i.e.*, redundant nodes) in the standard HSTs by BASE.

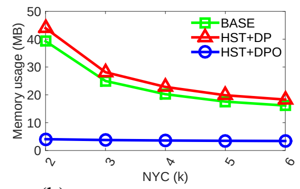
Results on scalability tests. Fig. 4g and Fig. 4h present the results of the scalability tests. Since BASE and HST+DP sometimes cannot be terminated in 7 days, we only show partial results of BASE and HST+DP. We can observe our algorithm HST+DPO is the most efficient in terms of both running time and memory usage. For example, HST+DPO is up to $43.8\times$ faster than the baseline BASE and it needs $11.4\times$ less spaces than HST+DP.

B. Experiments on Insertion

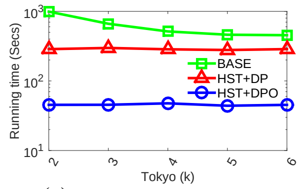
1) Experimental Setup: **Datasets.** We use the aforementioned real and synthetic datasets, where default settings are marked by bold in Table II and Table III. In each test case, we randomly sample 80% of the total points as the predefined points and fix the number of insertions as 10. We sample 1%-



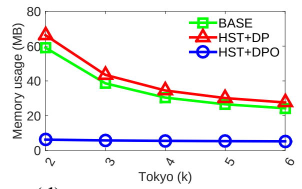
(a) Running time on *NYC*



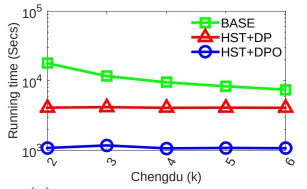
(b) Memory usage on *NYC*



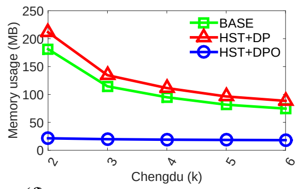
(c) Running time on *Tokyo*



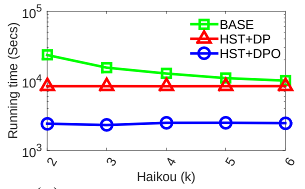
(d) Memory usage on *Tokyo*



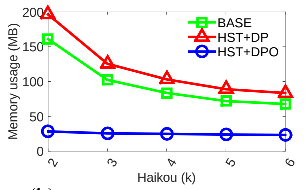
(e) Running time on *Chengdu*



(f) Memory usage on *Chengdu*

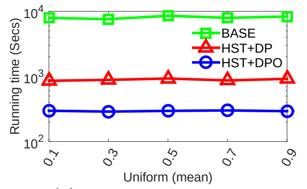


(g) Running time on *Haikou*

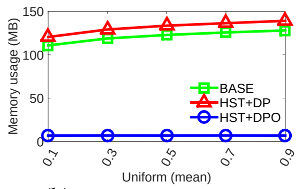


(h) Memory usage on *Haikou*

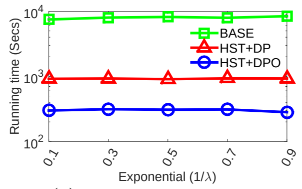
Fig. 3: Results of construction on real datasets of *NYC*, *Tokyo*, *Chengdu* and *Haikou*



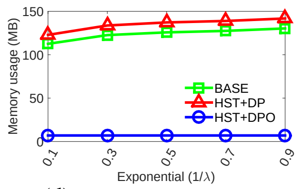
(a) Running time on *Uni*



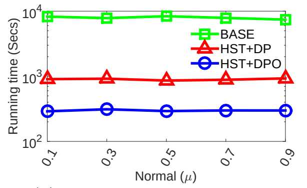
(b) Memory usage on *Uni*



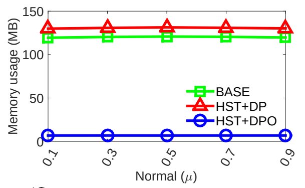
(c) Running time on *Exp*



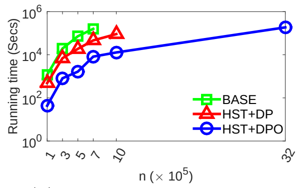
(d) Memory usage on *Exp*



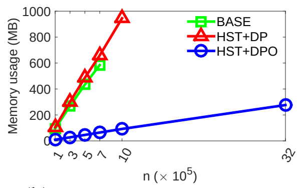
(e) Running time on *Nor* (μ)



(f) Memory usage on *Nor* (μ)



(g) Running time (scalability)



(h) Memory usage (scalability)

Fig. 4: Results of construction on synthetic datasets and scalability tests

3% of the remaining points to be inserted for each time and all the remaining points will be inserted at the last time.

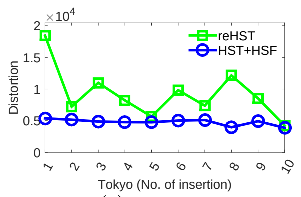
Compared Algorithms. We compare our method HST+HSF with the baseline of reconstructing the HST (denoted by reHST). They both use HST+DPO to construct an HST. The parameter c in HST+HSF is set to 5.

Metrics. We evaluate these algorithms in terms of *distortion*, *running time* and *memory usage*, where *distortion* is calculated based on the newly inserted points (*i.e.*, the maximum stretch of the distances between the newly inserted points and the current points), *running time* is the time cost for each insertion, and *memory usage* is the space cost of the embedding.

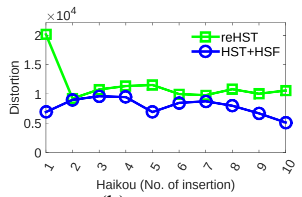
The **implementation** is same as the previous experiments.

2) Experimental Results: Fig. 5 and Fig. 6 illustrate the experimental results. Since we fix the number of insertions as 10, $x = 1, \cdots, 10$ in the horizontal axis denotes the number of insertions. Due to space limitation, we omit the experimental results on *NYC* and *Chengdu*, which have similar patterns with the results on *Tokyo* and *Haikou*, respectively.

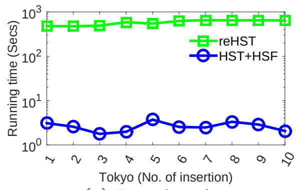
Results on real datasets. The first column of Fig. 5 presents the results of *Tokyo*. In terms of *distortion*, our algorithm HST+HSF always outperforms the baseline reHST. For example, the distortion of HST+HSF is up to $3.4\times$ smaller than reHST. In Fig. 5a, there are fluctuations in the results since the inserted points are sampled in a purely random way. As for *running time*, HST+HSF is $146\times$-$310\times$ faster than reHST.



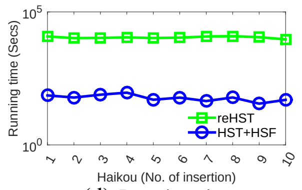
(a) Distortion



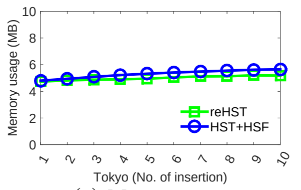
(b) Distortion



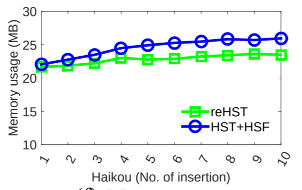
(c) Running time



(d) Running time



(e) Memory usage

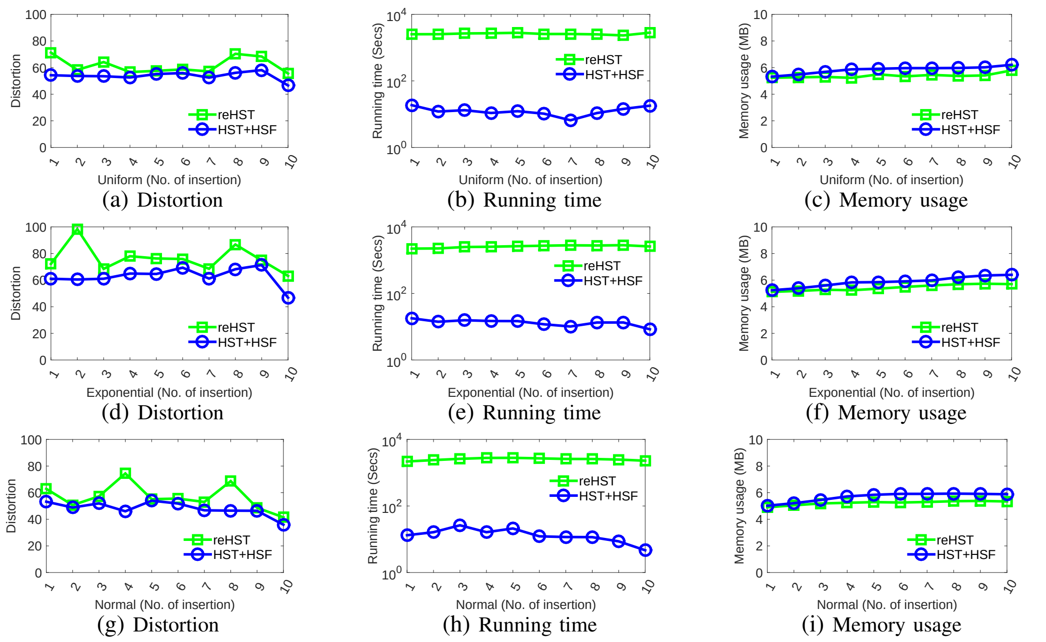


(f) Memory usage

Fig. 5: Results of insertion on real datasets

The baseline will be notably inefficient when insertions are frequently occurred. As for *memory usage*, HST+HSF needs at most 0.5MB more spaces than reHST due to the cost of extra arrays (*e.g.*, NN and NND).

The last column of Fig. 5 illustrates the results of *Haikou*.



(a) Distortion (b) Running time (c) Memory usage
(d) Distortion (e) Running time (f) Memory usage
(g) Distortion (h) Running time (i) Memory usage

Fig. 6: Results of insertion on synthetic datasets under Uniform, Exponential and Normal distributions

In Fig. 5b, our algorithm is much more effective, since the *distortions* of HST+HSF are constantly smaller than the distortions of reHST. In terms of *running time*, HST+HSF is still notably better than reHST. For example, HST+HSF is up to 317× faster than reHST. As for *memory usage*, both algorithms are efficient (*e.g.*, less than 26MB). Similar to the previous results, HST+HSF consumes slightly more spaces.

Results on synthetic datasets. Fig. 6 illustrates the results on synthetic datasets under Uniform (first row), Exponential (second row) and Normal distribution (last row), respectively. Under all these three distributions, our algorithm HST+HSF always outperforms the baseline reHST in terms of effectiveness (*i.e.*, *distortion*). As for *running time*, HST+HSF is up to 391×, 308× and 491× faster than reHST in Fig. 6b, Fig. 6e and Fig. 6h, respectively. In terms of *memory usage*, both algorithms consume less than 6.5MB, since they both apply our optimization techniques in the construction of HSTs. Our algorithm HST+HSF needs more spaces since it maintains extra arrays like NN and NND.

C. Experimental Summary

We summarize our experimental findings as follows.

In the experiments of *construction*, our method HST+DPO is always the most efficient. For example, HST+DPO is up to 24.7× and 29.8× faster than the state-of-the-art method BASE in the real datasets and synthetic datasets, respectively. Moreover, 85% of the baseline's spaces can be saved by HST+DPO. The comparisons with HST+DP demonstrate that our optimization techniques (*e.g.*, improved DP strategy, compressing strategy and pruning strategy) are effective.

In the experiments of *insertion*, our proposed algorithm HST+HSF outperforms the baseline reHST in terms of effectiveness (*e.g.*, by up to 3.4× better). HST+HSF is more efficient than reHST by up to 491× shorter *running time*.

VII. Related Work

Metric Embedding was first proposed in the 1980s. Specifically, Johnson and Lindenstrauss [40] focus on embedding metric spaces into a Hilbert space. Alon *et al.* [41] study the embeddings into a spanning tree. Among these metric embeddings, *Hierarchically Separated Tree* (HST) is one of the most prevalent data structure, which was first proposed by Bartal [8]. To minimize the *distortion* of the HST, existing studies [7]–[10], [20], [26], [42] mainly focus on improving the theoretical guarantees of the (expected) distortion.

Specifically, Bartal [9] proved that the expected distortion of the HST is $O(k \log n \log\log n)$. Konjevod *et al.* [42] further improved the guarantee to $O(\log \Delta)$, where Δ denotes the diameter of the original metric space. Gao *et al.* [20] studies how to minimize the communication cost of this method in a sensor network. Indyk [7] converted a quadtree into the HST with an approximation ratio of $O(\log^4 n)$. Among the existing studies, Fakcharoenphol *et al.* [10], [26] proposed the state-of-the-art construction method (*i.e.*, Algorithm 1) with the tight distortion guarantee ($O(\log n)$). Blelloch *et al.* [27] propose a parallel construction method with the same guarantee.

Although HST has been widely used in many existing studies, much less attention has been paid in the *efficiency*. For instance, the construction method in [10], [26] takes high complexity (*e.g.*, $O(n^3)$ time) in the *worst-case* and the construction method in [27] takes $O(n^2 \log n)$ time in the *average-case*. However, their complexities are not optimal (*e.g.*, $O(n^2)$ time in the *worst-case*). Moreover, existing studies usually assumed the geometric data was static and hence they did not provide flexible solutions to the insertion operation.

VIII. Conclusion

In this paper, we study the *<u>E</u>mbedding <u>A</u>rbitrary metrics by <u>T</u>ree metrics* (EAT) problem and focus on improving the efficiency of a well-known index called Hierarchically Separated Tree (HST). Specifically, the state-of-the-art method still requires high time complexity ($O(n^3)$) and space cost ($O(n^2)$) to construct an HST. Moreover, existing studies have no efficient support in insertions, which can be occurred in real applications. To address these limitations, we have proposed a new construction method with the optimal time and space complexity. It improves the time complexity from $O(n^3)$ to $O(n^2)$ and reduces the space cost from $O(n^2)$ to $O(n)$. To flexibly support insertion, we have introduced a new data structure called Hierarchically Separated Forest (HSF), *i.e.*, a collection of HSTs. Based on an HSF, we have designed an efficient insertion algorithm with a tight guarantee in the distortion ($O(\log n)$). Finally, we have conducted extensive experiments on both real and synthetic datasets. The experimental results validate the superior performance of our methods in the effectiveness and running time.

Acknowledgment

This work is partially supported by National Key Research and Development Program of China Grant No. 2018AAA0101100, the Hong Kong RGC GRF Project 16207617, CRF Project C6030-18G, C1031-18G, C5026-18G, AOE Project AoE/E-603/18, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants ITS/044/18FX and ITS/470/18FX, Microsoft Research Asia Collaborative Research Grant, Didi-HKUST joint research lab project, and Wechat and Webank Research Grants. Yongxin Tong's work is partially supported by the National Science Foundation of China (NSFC) under Grant No. 61822201 and U1811463. Yongxin Tong and Lei Chen are the corresponding authors in this paper.

References

[1] R. H. Güting, "Geo-relational algebra: A model and query language for geometric database systems," in *EDBT*, 1988, pp. 506–527.

[2] H. Six and P. Widmayer, "Spatial searching in geometric databases," in *ICDE*, 1988, pp. 496–503.

[3] R. H. Güting, "Gral: An extensible relational database system for geometric applications," in *VLDB*, 1989, pp. 33–44.

[4] S. Har-Peled, *Geometric approximation algorithms*. American Mathematical Society, 2011, no. 173.

[5] C. D. Toth, J. O'Rourke, and J. E. Goodman, *Handbook of discrete and computational geometry*. Chapman and Hall/CRC, 2017.

[6] D. P. Williamson and D. B. Shmoys, *The design of approximation algorithms*. Cambridge university press, 2011.

[7] P. Indyk, "Algorithmic applications of low-distortion geometric embeddings," in *FOCS*, 2001, pp. 10–33.

[8] Y. Bartal, "Probabilistic approximations of metric spaces and its algorithmic applications," in *FOCS*, 1996, pp. 184–193.

[9] Y. Bartal, "On approximating arbitrary metrices by tree metrics," in *STOC*, 1998, pp. 161–168.

[10] J. Fakcharoenphol, S. Rao, and K. Talwar, "A tight bound on approximating arbitrary metrics by tree metrics," in *STOC*, 2003, pp. 448–455.

[11] A. Backurs, P. Indyk, K. Onak, B. Schieber, A. Vakilian, and T. Wagner, "Scalable fair clustering," in *ICML*, 2019, pp. 405–413.

[12] B. Behsaz, Z. Friggstad, M. R. Salavatipour, and R. Sivakumar, "Approximation algorithms for min-sum k-clustering and balanced k-median," *Algorithmica*, vol. 81, no. 3, pp. 1006–1030, 2019.

[13] Y. Tong, J. She, B. Ding, L. Chen, T. Wo, and K. Xu, "Online minimum matching in real-time spatial data: Experiments and analysis," *PVLDB*, vol. 9, no. 12, pp. 1053–1064, 2016.

[14] Z. Chen, P. Cheng, Y. Zeng, and L. Chen, "Minimizing maximum delay of task assignment in spatial crowdsourcing," in *ICDE*, 2019, pp. 1454–1465.

[15] Y. Tong, Z. Zhou, Y. Zeng, L. Chen, and C. Shahabi, "Spatial crowdsourcing: a survey," *The VLDB Journal*, vol. 29, no. 1, pp. 217–250, 2020.

[16] Y. Zeng, Y. Tong, and L. Chen, "Last-mile delivery made practical: An efficient route planning framework with theoretical guarantees," *PVLDB*, vol. 13, no. 3, pp. 320–333, 2019.

[17] C. Coester and E. Koutsoupias, "The online k-taxi problem." in *STOC*, 2019, pp. 1136–1147.

[18] Y. Esencayi, M. Gaboardi, S. Li, and D. Wang, "Facility location problem in differential privacy model revisited," in *NIPS*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 8489–8498.

[19] Q. Tao, Y. Tong, Z. Zhou, Y. Shi, L. Chen, and K. Xu, "Differentially private online task assignment in spatial crowdsourcing: A tree-based approach," in *ICDE*, 2020, pp. 517–528.

[20] J. Gao, L. J. Guibas, N. Milosavljevic, and D. Zhou, "Distributed resource management and matching in sensor networks," in *IPSN*, 2009, pp. 97–108.

[21] Y. Yu, B. Krishnamachari, and V. P. Kumar, *Information processing and routing in wireless sensor networks*. World Scientific, 2006.

[22] J. Li, A. Deshpande, and S. Khuller, "Minimizing communication cost in distributed multi-query processing," in *ICDE*, 2009, pp. 772–783.

[23] S. Gollapudi and D. Sivakumar, "Framework and algorithms for trend analysis in massive temporal data sets," in *CIKM*, 2004, pp. 168–177.

[24] M. Cygan, A. Czumaj, M. Mucha, and P. Sankowski, "Online facility location with deletions," in *ESA*, 2018, pp. 21:1–21:15.

[25] Y. Azar and N. Touitou, "General framework for metric optimization problems with delay or with deadlines," in *FOCS*, 2019, pp. 11–22.

[26] J. Fakcharoenphol, S. Rao, and K. Talwar, "A tight bound on approximating arbitrary metrics by tree metrics," *Journal of Computer and System Sciences*, vol. 69, no. 3, pp. 485–497, 2004.

[27] G. E. Blelloch, A. Gupta, and K. Tangwongsan, "Parallel probabilistic tree embeddings, k-median, and buy-at-bulk network design," in *SPAA*, 2012, pp. 205–213.

[28] Y. Zeng, Y. Tong, and L. Chen, "An efficient index for embedding arbitrary metric spaces," 2020, http://home.cse.ust.hk/%7Eyzengal/hst.pdf.

[29] D. Yang, D. Zhang, V. W. Zheng, and Z. Yu, "Modeling user activity preference by leveraging user spatial temporal characteristics in lbsns," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 45, no. 1, pp. 129–142, 2015.

[30] "Foursquare." [Online]. Available: https://foursquare.com/

[31] "GAIA initiative." [Online]. Available: http://gaia.didichuxing.com

[32] "Didi Chuxing." [Online]. Available: http://www.didichuxing.com/

[33] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu, "A unified approach to route planning for shared mobility," *PVLDB*, vol. 11, no. 11, pp. 1633–1646, 2018.

[34] Y. Zeng, Y. Tong, Y. Song, and L. Chen, "The simpler the better: An indexing approach for shared-route planning queries," *PVLDB*, vol. 13, no. 13, pp. 3517–3530, 2020.

[35] Y. Zeng, Y. Tong, L. Chen, and Z. Zhou, "Latency-oriented task completion via spatial crowdsourcing," in *ICDE*, 2018, pp. 317–328.

[36] Q. Tao, Y. Zeng, Z. Zhou, Y. Tong, L. Chen, and K. Xu, "Multi-worker-aware task planning in real-time spatial crowdsourcing," in *DASFAA*, 2018, pp. 301–317.

[37] Y. Li, J. Fang, Y. Zeng, B. Maag, Y. Tong, and L. Zhang, "Two-sided online bipartite matching in spatial data: experiments and analysis," *GeoInformatica*, vol. 24, no. 1, pp. 175–198, 2020.

[38] B. Zhao, P. Xu, Y. Shi, Y. Tong, Z. Zhou, and Y. Zeng, "Preference-aware task assignment in on-demand taxi dispatching: An online stable matching approach," in *AAAI*, 2019, pp. 2245–2252.

[39] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *SIGMOD*, 2016, pp. 1071–1085.

[40] W. B. Johnson and J. Lindenstrauss, "Extensions of lipschitz mappings into a hilbert space," *Contemporary mathematics*, vol. 26, no. 189-206, p. 1, 1984.

[41] N. Alon, R. M. Karp, D. Peleg, and D. West, "A graph-theoretic game and its application to the k-server problem," *SIAM Journal on Computing*, vol. 24, no. 1, pp. 78–100, 1995.

[42] G. Konjevod, R. Ravi, and F. S. Salman, "On approximating planar metrics by tree metrics," *Information Processing Letters*, vol. 80, no. 4, pp. 213–219, 2001.

[43] A. Meyerson, A. Nanavati, and L. J. Poplawski, "Randomized online algorithms for minimum metric bipartite matching," in *SODA*, 2006, pp. 954–959.

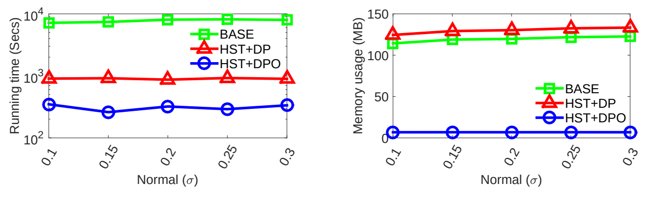
[44] N. Bansal, N. Buchbinder, A. Gupta, and J. Naor, "An O ($\log^2$ k)-competitive algorithm for metric bipartite matching," in *ESA*, 2007, pp. 522–533.

[45] N. Bansal, N. Buchbinder, A. Gupta, and J. Naor, "A randomized o(log2 k)-competitive algorithm for metric bipartite matching," *Algorithmica*, vol. 68, no. 2, pp. 390–403, 2014.

[46] A. Meyerson, A. Nanavati, and L. J. Poplawski, "Randomized online algorithms for minimum metric bipartite matching," in *SODA*, 2006, pp. 954–959.

Algorithm 4: Deletion

```
input : a $k$-HSF $\mathcal{F}$ and the deleted points $\mathcal{V}^-$
1 Mark the points in $\mathcal{V}^-$ as deleted;
2 foreach point $x$ whose nearest neighbor in $\mathcal{V}_i$ has just been marked do
3     Update $\mathsf{NN}[x][i]$ and $\mathsf{NND}[x][i]$ by selecting the nearest neighbor in the remaining points;
```



(a) Running time on *Nor* (σ) (b) Memory usage on *Nor* (σ)

Fig. 7: Results of construction on synthetic datasets under Normal distributions (when varying the parameter σ)

Appendix A Efficient Deletion Algorithm

When deleting some points, our **basic idea** is based on the lazy deletion. Specifically, we first mark the leaves of these points as deleted, and then update NN and NND of the points in the other HSTs whose nearest neighbors have just been deleted. The marked points will be actually deleted when their located HSTs are re-constructed.

Algorithm Details. Algorithm 4 illustrates the detailed procedure. In line 1, we mark the points in $\mathcal{V}^-$ as deleted. In lines 2-3, we update both NN and NND for each point whose nearest neighbor has just been deleted.

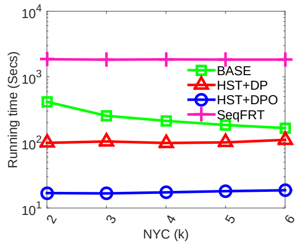
Time Complexity. Algorithm 3 takes $O(cm) \sim O(m)$ time in lines 1. Lines 2-3 take $O(\mathsf{rnn} \cdot |\mathcal{V}^-|)$ time to update the nearest neighbors, where rnn denotes the maximum number of reverse nearest neighbor for each deleted point. Thus, the time complexity of deleting points $\mathcal{V}^-$ is $O(\mathsf{rnn}|\mathcal{V}^-|)$.

Appendix B Experimental Results of Normal Distribution (σ)

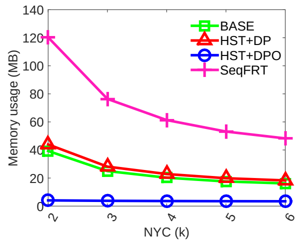
This section presents the experimental results on the synthetic datasets that follow the Normal distribution and vary the value of σ. Specifically, Fig. 7 shows the results on the synthetic datasets of Normal distribution (when varying the standard deviation σ). In Fig. 7a and Fig. 7b, HST+DPO is always the most efficient construction method in terms of time cost and space consumption. BASE is slower than HST+DP while BASE takes less spaces than HST+DP. Overall, our construction algorithm HST+DPO is $20.5\times$-$28.5\times$ faster than BASE and also consumes $16.7\times$-$18.0\times$ spaces than BASE.

Appendix C Experimental Comparisons with SeqFRT

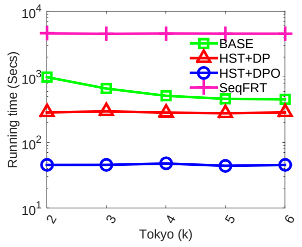
In this section, we use SeqFRT to denote the proposed algorithm in [27]. Here, we compare with the sequential version of SeqFRT instead of its parallel implementation in



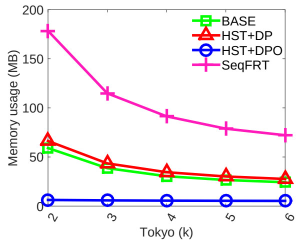
(a) Running time on *NYC*



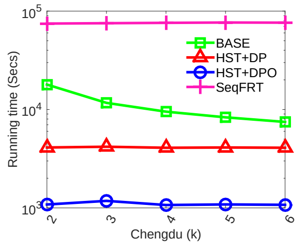
(b) Memory usage on *NYC*



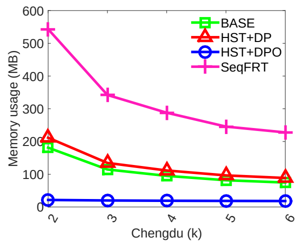
(c) Running time on *Tokyo*



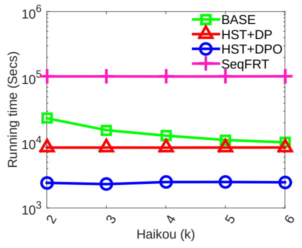
(d) Memory usage on *Tokyo*



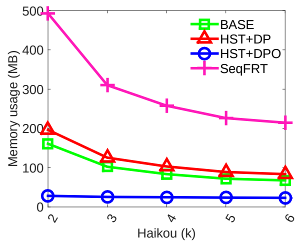
(e) Running time on *Chengdu*



(f) Memory usage on *Chengdu*

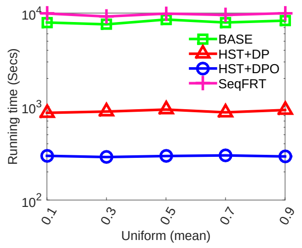


(g) Running time on *Haikou*

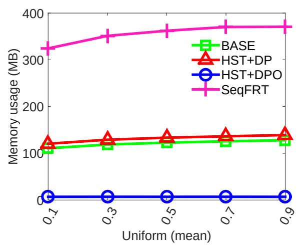


(h) Memory usage on *Haikou*

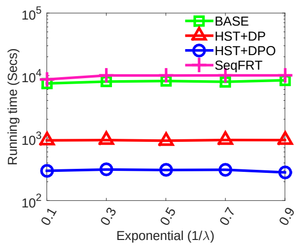
Fig. 8: Results of construction on real datasets of *NYC*, *Tokyo*, *Chengdu* and *Haikou*



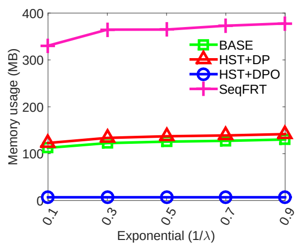
(a) Running time on *Uni*



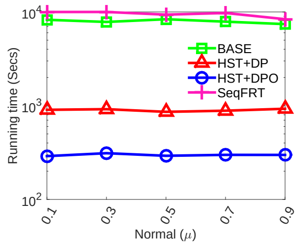
(b) Memory usage on *Uni*



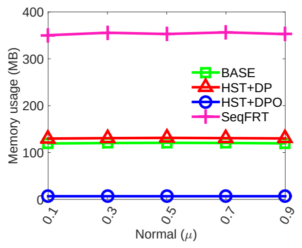
(c) Running time on *Exp*



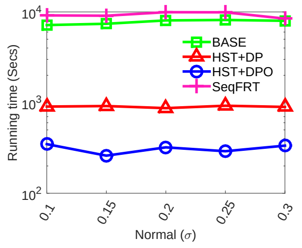
(d) Memory usage on *Exp*



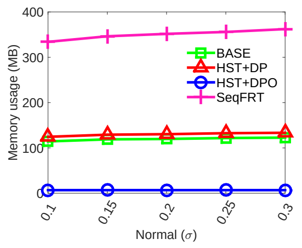
(e) Running time on *Nor* (μ)



(f) Memory usage on *Nor* (μ)



(g) Running time on *Nor* (σ)



(h) Memory usage on *Nor* (σ)

Fig. 9: Results of construction on synthetic datasets under Uniform, Exponential and Normal distributions

[27], since our experiments mainly focus on the sequential versions of the construction methods.

Fig. 8 presents the experimental results on real datasets (*NYC*, *Tokyo*, *Chengdu* and *Haikou*). We can observe that SeqFRT is always less efficient than all the other algorithms in these datasets. For instance, SeqFRT is $4\times$-$11\times$ slower than BASE and consumes $2\times$-$3\times$ more spaces than BASE at the same time. Compared with our algorithm HST+DPO, SeqFRT is $40\times$-$109\times$ slower and takes up to $29\times$ more spaces. We can also observe that the time cost of SeqFRT is not significantly changed with the increase of k. This is because the *average-case* time complexity of SeqFRT is $O(n^2 \log n)$, which is independent of the height of the HST (*i.e.*, $\mathcal{H} = \log_k \Delta$). The results also demonstrate that BASE is a more suitable baseline than SeqFRT in terms of the efficiency.

Fig. 9 presents the experimental results on synthetic datasets under different distributions (Uniform, Exponential and Normal distributions). We can also observe that SeqFRT is always the least efficient among all the compared algorithms. Since SeqFRT is less efficient than our baseline BASE, it is reasonable to select BASE instead of SeqFRT as the baseline in the experiments of constructions.

Appendix D
A Case Study on Metric Bipartite Matching

In this section, we demonstrate that our technique in Sec. V are helpful to break the impractical assumption in real-world problems. Here, the impractical assumption is that

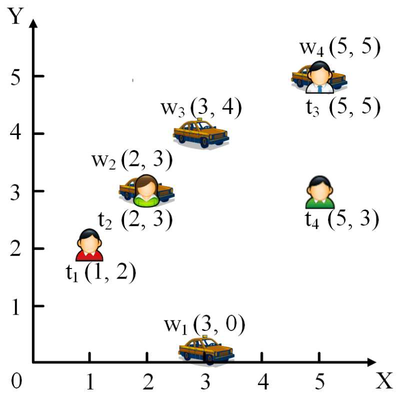


Fig. 10: Locations of left-hand vertices (W) and right-hand vertices (T) [13]

all objects are assumed to be known in advance. Specifically, we first present the Online Minimum Metric Bipartite Matching (OMMBM) problem in Appendix D-A, and then introduce two HST-based solutions (*i.e.*, HST-Greedy and HST-Reassignment) to this problem in Appendix D-B and Appendix D-C. Finally, we show how to break above assumption by our technique without asymptotically changing theoretical guarantees in Theorem 5 and Theorem 6.

A. Online Metric Bipartite Matching Problem

The definition of Online Minimum Metric Bipartite Matching (OMMBM) problem is as follows.

Definition 10 (OMMBM Problem): Given a set of left-hand vertices W with specific locations, a set of right-hand vertices T whose locations are unknown before they appear, and a metric distance function $\mathcal{D}(.,.)$ in a metric space, the OMMBM problem is to find a matching M to minimize the total distance $Cost(M) = \sum_{(t,w)\in M} \mathcal{D}(t,w)$ between the matched pairs such that the following constraints are satisfied:

- Real-time constraint: once a right-hand vertex appears, a left-hand vertex must be immediately allocated to it before the next right-hand vertex appears.
- Invariable constraint: once a left-hand vertex is allocated to a right-hand vertex, the allocation cannot be revoked.
- Cardinality constraint: $k = |M| = \min\{|T|, |W|\}$, where $|.|$ is the size of a given set.

We illustrate the OMMBM problem by the example below.

Example 8: Suppose a taxi dispatching platform has four service providers $w_1 - w_4$ (*i.e.*, the left-hand vertices) and four taxi-calling tasks $t_1 - t_4$ (*i.e.*, the right-hand vertices) from users. The locations of the taxis and users (revealed as they arrive) are labelled in a two-dimensional Euclidean space in Fig. 10. The platform wants to minimize the overall travel distance cost, *e.g.*, Euclidean distance, for the assigned taxis to pick up the users. The taxis are assumed to be relatively static in a time interval (*e.g.*, 5 minutes) and hence their locations are known in advance, while the users dynamically appear.

In the offline scenario, where all locations are known, the optimal matching is $\{(t_1, w_1), (t_2, w_2), (t_3, w_4), (t_4, w_3)\}$ with a cost of $2\sqrt{2} + \sqrt{5} \approx 5.06$. However, in the online scenario,

TABLE IV: Arrival orders of four right-hand vertices

Arrival Order	1st	2nd	3rd	4th
1st Order	t_1	t_2	t_3	t_4
2nd Order	t_3	t_4	t_2	t_1

a taxi (*i.e.*, left-hand vertex) should be immediately allocated to each newly arrived task (*i.e.*, right-hand vertex) in the online scenarios. For example, we consider the simple nearest neighbor strategy, which allocates each new arrival user to its currently nearest unmatched taxi. For the "2nd order" in Table IV, the matching returned by this strategy is the same as the offline optimal matching. However, for the "1st order", the cost of Greedy is 6.43, which is worse than that of the offline optimal matching.

To evaluate the theoretical guarantees of solutions to the OMMBM problem, we introduce the standard analysis model called *competitive analysis model* in the following.

Definition 11 (Competitive Ratio (CR)): The competitive ratio of an online algorithm in the adversarial model for the OMMBM problem is defined as follows:

$$CR_A = \max_{\forall G(T,W) \textit{ and } \forall \sigma \textit{ of } T} \frac{Cost(M)}{Cost(OPT)} \tag{8}$$

where $G(T, W)$ is an arbitrary metric bipartite graph of left-hand vertices and right-hand vertices, where the weight of an edge in the $G(T, W)$ corresponds to the distance between the two objects in T and W respectively, σ is an arbitrary arrival order of the right-hand vertices in T, $Cost(M)$ is the total distance cost generated by the online algorithm, and $Cost(OPT)$ is the optimal total distance cost in the offline scenario.

B. The HST-Greedy Algorithm

To address the OMMBM problem, Meyerson *et al.* [43] first devise the matching algorithm on HST, *i.e.*, HST-Greedy. The **main idea** is to adopt the matching strategy of Greedy on the HST metric.

Specifically, HST-Greedy first builds an α-HST by Algorithm 2 for the left-hand vertices, where all the service providers are projected into a tree metric. For each newly arrived right-hand vertex, HST-Greedy applies the following two steps as the matching policy: (1) HST-Greedy first finds the left-hand vertex v_i currently nearest to t_i in the original 2D space. (2) HST-Greedy then chooses an unmatched left-hand vertex w_i nearest to v_i on the HST metric. If there are multiple left-hand vertices that have the same distance to v_i on the tree metric, the algorithm randomly chooses one as w_i. If v_i is also an unmatched left-hand vertex, w_i is replaced by v_i to be matched to t_i. Otherwise, w_i is directly matched to t_i. Thus, the pair (t_i, w_i) is added to the final online matching. **Notice, the first step is based on our technique in Sec. V, where the HSF has two HSTs: one is built upon W and the other is built upon T. Since a match to each $t_i \in T$ should be immediately decided when t_i appears, we do not need to actually construct the HST of T.** The detailed procedure is illustrated with the following example.

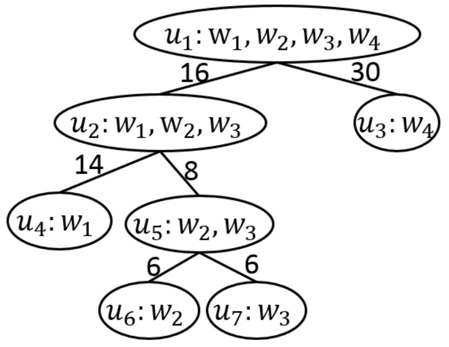


Fig. 11: A 2-HST constructed for HST-Greedy and HST-Reassignment by our Algorithm 2 in this paper

Example 9: Back to our running example in Example 8, we assume that users dynamically arrived following the "1st order" and an HST is constructed as shown in Fig. 11. Thus, when right-hand vertex t_1 arrives, HST-Greedy first finds the left-hand vertex w_2 who is currently nearest to him/her in the first step. In the second step of HST-Greedy, HST-Greedy matches the left-hand vertex w_2 to right-hand vertex t_1. Similarly, when the right-hand vertex t_2 arrives, HST-Greedy finds the left-hand vertex w_2 in the first step and matches left-hand vertex w_3 to right-hand vertex t_2 in the second step. Finally, the matching result of HST-Greedy is $\{(t_1, w_2), (t_2, w_3), (t_3, w_4), (t_4, w_1)\}$ with the cost of 6.43.

In [43], with the α-HST structure, the total cost of the greedy-based matching strategy on the HST metric is $\mathcal{O}(\log k)$ when $\alpha > 2 \ln k + 1$. In addition, α-HST can also guarantee that the expectation of the distance of two vertices on the tree is no greater than $\mathcal{O}(\alpha \log k)$ times the distance in the original metric. Therefore, the final competitive ratio of HST-Greedy in [43] is $\mathcal{O}(\log^3 k)$.

However, the implementation of HST-Greedy in this subsection is *slightly different* with the proposed algorithm in [43]. Since [43] **assumes** that all the locations of users is a subset of vertices in the original metric space, *i.e.*, $\{\text{location of } t | t \in T\} \subseteq V$. This assumption may be not established in realistic since users can be anywhere in practice, which indicates that the size of V can be arbitrarily large. Thus, in the first step of greedy-based matching strategy, HST-Greedy in [43] instead finds the exact vertex in HST which corresponds to the location of the right-hand vertex. To break this unrealistic assumption, our implementation constructs the HST *only* based on the locations of the left-hand vertices, since such information is known in advance. Therefore, we can find the nearest left-hand vertex for each newly arrived right-hand vertex in the first step, which is the only difference with the matching strategy on HST in [43]. In the following, we prove that the competitive ratio of our implementation is the same as [43] without any assumption about the online users.

Theorem 5: Our implementation of HST-Greedy also achieves the competitive ratio of $\mathcal{O}(\log^3 k)$ without any assumption about the locations of online users.

Proof: Let I be any instance of the OMMBM problem. To ease of presentation, we abuse notations w and t to be the location of left-hand vertex w and right-hand vertex t, respectively. We transform this instance into I' by only replacing t with the location of his/her nearest left-hand vertex (denoted by $nn(t)$). Then in the first step of HST-Greedy, our implementation will find the vertex on HST which corresponds to the exact location of any right-hand vertex of the instance I'. Let M'^* be the optimal matching of instance I' and M' be the matching result of instance I' based on our implementation. It is obvious that our HST-Greedy will obtain the same matching result on the instance I due to the transformation of instance I'. According to the theoretical result (Theorem 3.1) in [43], we have

$$\begin{aligned} Cost(M') &= \sum_{(t,w)\in M} \mathcal{D}(nn(t), w) \\ Cost(M'^*) &= \sum_{(t,w)\in M'^*} \mathcal{D}(nn(t), w) \\ \mathbb{E}[Cost(M')] &\le \mathcal{O}(\log^3 k) \times Cost(M'^*) \end{aligned}$$

Thus, we can bound the total cost of matching result M' of the original instance I as follows.

$$\begin{aligned} Cost(M) &= \sum_{(t,w)\in M} \mathcal{D}(t, w) \\ &\le \sum_{(t,w)\in M} \big(\mathcal{D}(t, nn(t)) + \mathcal{D}(nn(t), w)\big) \\ &\le \Big(\sum_{t\in T} \mathcal{D}(t, nn(t))\Big) + \Big(\sum_{(t,w)\in M} \mathcal{D}(nn(t), w)\Big) \\ &\le Cost(M^*) + Cost(M') \end{aligned}$$

We also need to bound the total cost of optimal matching result M'^* of the transformed instance I' as follows.

$$\begin{aligned} Cost(M'^*) &= \sum_{(t,w)\in M'^*} \mathcal{D}(nn(t), w) \\ &\le \sum_{(t,w)\in M^*} \mathcal{D}(nn(t), w) \\ &\le \sum_{(t,w)\in M^*} \big(\mathcal{D}(nn(t), t) + \mathcal{D}(t, w)\big) \\ &\le \Big(\sum_{t\in T} \mathcal{D}(t, nn(t))\Big) + \Big(\sum_{(t,w)\in M^*} \mathcal{D}(t, w)\Big) \\ &\le 2 \times Cost(M^*) \end{aligned}$$

Finally, we can infer the competitive ratio of our extended implementation, which is the same as [43].

$$\begin{aligned} \mathbb{E}[Cost(M)] &\le Cost(M^*) + \mathbb{E}[Cost(M')] \\ &\le Cost(M^*) + \mathcal{O}(\log^3 k) \times Cost(M'^*) \\ &\le Cost(M^*) + \mathcal{O}(\log^3 k) \times 2Cost(M^*) \\ &\le \mathcal{O}(\log^3 k) \times Cost(M^*) \end{aligned}$$

■

C. The HST-Reassignment Algorithm

To achieve better competitive ratio, Bansal *et al.* devise a more complex matching algorithm called HST-Reassignment on HST [44], [45]. Its **main idea** is find a match for each newly arrived right-hand vertex based on the following strategies: (1) They first obtain an optimal matching result based on currently arrived users; (2) Due to the invariable constraint, they further change the optimal matching, which re-assigns the left-hand vertices, into a feasible matching where no allocation is revoked. The difference is that Bansal *et al.* do not use the existing offline matching algorithm (*e.g.*, Hungarian algorithm). Instead, they utilize the aforementioned properties of HST and devise a level-by-level algorithm which obtains the optimal matching.

Specifically, when a new right-hand vertex t_i appears, the algorithm applies the following two steps as their matching policy: (1) It first gets an optimal matching based on currently arrived users by Algorithm 5 (which will be explained later) on the HST. (2) Since this matching must have an unmatched left-hand vertex (denoted by w_i^*), the algorithm will assign the left-hand vertex w_i^* to the new right-hand vertex t_i.

In Algorithm 5, the main idea is to match a left-hand vertex in the level-by-level manner. Specifically, if the algorithm traverse a vertex on HST which contains a left-hand vertex w assigned to right-hand vertex t, let L_w and L_t be the level of the vertices. In lines 1-2, L_w and L_t are initialized only once with the value of $-\infty$ and the level of leaf since they are not matched yet. In each iteration, the algorithm first finds the nearest left-hand vertex v to right-hand vertex t (line 5) and then finds the highest level $\mathcal{L} \leq L_t$ in which there exists an ancestor of v or itself such that it contains a left-hand vertex satisfying $\mathcal{L} > L_w$ (line 6). That is, level $\mathcal{L}$ is iterated in a level-by-level manner in line 6, *i.e.*, from the level of leaf to the level of root. In lines 7-8, we randomly select a left-hand vertex to the right-hand vertex t and update L_w and L_t with $\mathcal{L}$. Thus, the value of L_w is monotonically increasing since $\mathcal{L}$ must larger than L_w. The iteration will be terminated when an unmatched left-hand vertex in current online matching result has been found. In line 10, any right-hand vertex, who is not re-assigned, we just keep the previous allocations. We illustrate the whole procedure in the following.

Example 10: Back to our running example in Example 8. We assume that users dynamically arrived following the "1st order" and an HST is constructed as shown in Fig. 11. When the right-hand vertex t_1 appears, it first finds the nearest left-hand vertex w_2. In Algorithm 5, L_{t_1} and L_{w_2} are initialized with 4 and $-\infty$ respectively. We can easily find the lowest level $\mathcal{L} = 4$ ($\leq L_{t_1}$) because there exists w_2 in this level satisfying $\mathcal{L} > L_{w_2}$. After matching w_2 to t_1, both L_{t_1} and L_{w_2} change into $\mathcal{L} = 4$. Similarly, when the right-hand vertex t_2 appears, its nearest left-hand vertex is also w_2. Thus, we iteratively try the level $\mathcal{L}$ from 4 to 0 and the constraint $\mathcal{L} > L_{w_2}$ is not satisfied when $\mathcal{L} = 4$ or 3. However, when $\mathcal{L}$ equals to 2, there exists a left-hand vertex w_3 such that $\mathcal{L}$ is larger than $L_{w_3} = -\infty$. Thus, we find the highest $\mathcal{L}$, match w_3 to t_2 and change L_{w_3} and L_{t_2} into $\mathcal{L} = 2$. Finally, HST-Reassignment will match w_4 to t_3 and w_1 to t_4. Therefore, the final matching is $\{(t_1, w_2), (t_2, w_3), (t_3, w_4), (t_4, w_1)\}$ with the cost 6.43.

Algorithm 5: Adaptive Optimal Matching

```
Input: left-hand vertices $W$, newly arrived right-hand
       vertex $t_i \in T$ and current matching $M_i$
Output: an optimal matching $M_i^*$
1  Initialize $L_w$ with $-\infty$ only once for each $w \in W$;
2  Initialize $L_{t_i}$ with the level of leaf vertex;
3  $t \leftarrow t_i, M_i^* \leftarrow \emptyset$;
4  repeat
5      $v \leftarrow$ the nearest left-hand vertex to the right-hand
         vertex $t$ in the original metric;
6      Find the highest level $\mathcal{L} \leq L_t$ in which there exists
         an ancestor of $v$ or itself such that at least one of
         its contained left-hand vertices (denoted by $w$)
         satisfies $\mathcal{L} > L_w$;
7      Choose uniformly at random one left-hand vertex
         $w$ among all the satisfiable ones;
8      Match $w$ to $t$ in $M_i^*$ and set $L_w \leftarrow \mathcal{L}, L_t \leftarrow \mathcal{L}$;
9  until an unmatched left-hand vertex is assigned to $t$ in
     $M_i^*$;
10 Keep the assignment in $M_i$ for each right-hand vertex
     $t \in T_i$ who is not matched in $M_i^*$;
11 return $M_i^*$;
```

Different from HST-Greedy [46] that adopts an α-HST structure ($\alpha > 2\ln k + 1$), HST-Reassignment [45] only uses 2-HST structure, namely $\alpha = 2$. Even though the algorithm is named based on reassignment, the final matching still satisfies the invariable constraint. Instead, they only propose the restricted reasssignment model in their competitive analysis and they prove the algorithm achieves the competitive ratio of $\mathcal{O}(\log^2 k)$, which loses a logarithmic term than HST-Greedy. However, **HST-Reassignment also has the same unrealistic assumption as HST-Greedy**. Thus, we still apply our extension technique for HST-Greedy and only modify the implementation in line 5 of Algorithm 5, *i.e.*, we always find the nearest left-hand vertex to each right-hand vertex as his/her corresponding location. Note that in the proof of Theorem 5, we only use the matching result of HST-Greedy and do not use any property of the algorithm. It indicates that the proof is also established for our implementation of HST-Reassignment.

Theorem 6: Our implementation of HST-Reassignment also achieves the competitive ratio of $\mathcal{O}(\log^2 k)$ without any assumption about the locations of online users.

Proof: According to the theoretical results (Theorem 1 and Corollary 1) in [44], we have

$$Cost(M') = \sum_{(t,w)\in M} \mathcal{D}(nn(t), w) \tag{9}$$

$$Cost(M'^*) = \sum_{(t,w)\in M'^*} \mathcal{D}(nn(t), w) \tag{10}$$

$$\mathbb{E}[Cost(M')] \leq \mathcal{O}(\log^2 k) \times Cost(M'^*) \tag{11}$$

In Theorem 5, we have proved that

$$Cost(M) \leq Cost(M^*) + Cost(M') \tag{12}$$

$$Cost({M'}^*) \leq 2 \times Cost(M^*) \tag{13}$$

Thus, we can infer the competitive ratio of our extended implementation, which is the same as [44], [45].

$$\begin{aligned}
\mathbb{E}[Cost(M)] &= Cost(M^*) + \mathbb{E}[Cost(M')] \\
&\leq Cost(M^*) + \mathcal{O}(\log^2 k) \times Cost({M'}^*) \\
&\leq Cost(M^*) + \mathcal{O}(\log^2 k) \times 2Cost(M^*) \\
&\leq \mathcal{O}(\log^2 k) \times Cost(M^*)
\end{aligned}$$

■