

1 Dynamic Locality Sensitive Orderings in Doubling Metrics

2 An La ^{*} Hung Le[†]

3 Abstract

4 In their pioneering work, Chan, Har-Peled, and Jones (SICOMP 2020) introduced *locality-sensitive*
5 *ordering* (LSO), and constructed an LSO with a constant number of orderings for point sets in the
6 *d*-dimensional Euclidean space. Furthermore, their LSO could be made dynamic effortlessly under
7 point insertions and deletions, taking $O(\log(n))$ time per update by exploiting Euclidean geometry.
8 Their LSO provides a powerful primitive to solve a host of geometric problems in Euclidean spaces in
9 both dynamic and static settings. Filtser and Le (STOC 2022) constructed the first LSO with a constant
10 number of orderings in the more general setting of doubling metrics. However, their algorithm is
11 inherently static since it relies on several sophisticated constructions in intermediate steps, none
12 of which is known to have a dynamic version. Making their LSO dynamic would recover the full
13 generality of LSO and provide a general tool to dynamize a vast number of static constructions in
14 doubling metrics.

15 In this work, we give a dynamic algorithm that has $O(\log n)$ time per update for constructing an
16 LSO in doubling metrics under point insertions and deletions. To this end, we introduce a toolkit of
17 several new data structures: *a pairwise index tree* (PIT) which augments the standard net tree with
18 the pairwise property, *a pairwise tree cover* which in a certain sense is a tree counterpart of LSO, *a net*
19 *tree cover* for stabilizing the net tree, and *a leaf tracker* for keeping track of a DFS ordering of leaves
20 in a dynamic tree. A key technical problem that we solve in this work is stabilizing the dynamic
21 net tree of Cole and Gottlieb (STOC 2006), a central dynamic data structure in doubling metrics,
22 using a dynamic *net tree cover*. Specifically, we show that every update to the dynamic net tree can
23 be decomposed into a few very simple updates to trees in the net tree cover. As stability is the key to
24 any dynamic algorithm, our technique could be useful for other problems in doubling metrics.

25 We obtain several algorithmic applications from our dynamic LSO, including dynamic fault-tolerant
26 spanner, dynamic tree cover, dynamic nearest neighbor search with optimal search time, dynamic
27 (bichromatic) closest pair of points, all in doubling metrics. Most notably, we obtain the first dynamic
28 algorithm for maintaining an k -fault tolerant spanner in doubling metrics with optimal sparsity in
29 optimal $O(\log n)$ time per update.

^{*}University of Massachusetts Amherst. Email: anla@umass.edu.

[†]University of Massachusetts Amherst. Email: hungle@cs.umass.edu.

30 **Contents**

31	1	Introduction	2
32	1.1	Key Technical Ideas	4
33	1.2	Applications	10
34	2	Preliminaries	12
35	2.1	Basic Notation	12
36	2.2	Dynamic LSO from Dynamic Pairwise Tree Cover	12
37	3	Pairwise Tree Cover: Static Construction	13
38	3.1	The Static Construction	15
39	3.2	The Analysis	17
40	4	Pairwise Tree Cover: Dynamic Construction	21
41	4.1	Dynamic Net Tree Cover	25
42	4.1.1	Dynamic net tree cover construction	27
43	4.1.2	Analysis	29
44	4.2	Dynamic Pairing	32
45	5	Leaf Tracker	35
46	5.1	Special Case: Trees without Inactive Nodes	37
47	5.2	General Case: Maintaining DFS Ordering of a PIT	41
48	5.2.1	Updating σ	42
49	5.2.2	Active tracker data structure	43
50	6	Dynamic Net Tree	47
51	6.1	Rings	48
52	6.2	Containment Search and Internal Operations	49
53	6.3	Dynamic Net Tree Operations	54
54	6.4	Analysis	57
55	7	Applications of LSO	63
56	7.1	Dynamic VFT Spanners	63
57	7.2	Dynamic Tree Cover	64
58	7.3	Dynamic Closest Pair	66
59	7.4	Approximate Bichromatic Closest Pair	67
60	7.5	Dynamic Approximate Nearest Neighbors	68

61 1 Introduction

62 Chan, Har-Peled and Jones [10] introduced locality-sensitive ordering (LSO) as a powerful tool for
 63 solving geometric problems. Roughly speaking, an LSO of a point set S in a metric space (X, d_X) is a
 64 collection of linear orderings of points in S that has a *locality property*, namely, any two points $x, y \in X$
 65 are close in some ordering of the collection. Here being close means the points between x and y in the
 66 ordering are either close to x or to y .

67 **Definition 1** $((\tau, \varepsilon)\text{-LSO})$. Let S be a set of points in a metric space (X, d_X) . A collection of linear
 68 orderings, denoted by Σ is a (τ, ε) -locality sensitive ordering if:

- 69 • [Size.] Σ has at most τ ordering.
- 70 • [Covering.] there is a bijection between points in each ordering and S .
- 71 • [Locality.] for every $x, y \in S$, there exists an ordering $\sigma \in \Sigma$ such that any point z between x and y
 72 on σ is in distance $\varepsilon d_X(x, y)$ either from x or from y . That is, $\min\{d_X(z, x), d_X(z, y)\} \leq \varepsilon d_X(x, y)$.

73 One could think of an LSO as an “embedding” of S into a collection of lines so that geometric
 74 constructions for S in the (complicated and high dimensional) metric space (X, d_X) could be reduced to
 75 the 1-dimensional line. Therefore, LSO allows significant simplification of the constructions of many
 76 complicated objects, such as fault-tolerant spanners [10] and reliable spanners [5, 6, 16].

77 Chan, Har-Peled and Jones [10] constructed a (τ, ε) -LSO Σ for point sets in \mathbb{R}^d where the number of
 78 orderings $\tau = 2^{O(d)}\varepsilon^{-d}$. Thus, for a fixed ε and d , the number of orderings is a constant. Furthermore,
 79 their LSO could be easily made dynamic, since the construction is based on space partitioning. More
 80 specifically, for any given two points $p, q \in \mathbb{R}^d$, one could determine their relative positions in a given
 81 ordering $\sigma \in \Sigma$ — determine whether $p \prec_\sigma q$ or $q \prec_\sigma p$ — by applying bitwise operations on their
 82 coordinates. Therefore, one could represent each ordering in the LSO as a binary search tree, which
 83 supports point updates in $O(\log n)$ time per operation. The main takeaway here is that Euclidean
 84 geometry allows simple dynamization of their LSO.

85 Using their dynamic LSO in \mathbb{R}^d , Chan, Har-Peled and Jones [10] obtained a host of dynamic algorithms
 86 for geometric problems, such as dynamic bichromatic closest pair of points, dynamic spanners, dynamic
 87 vertex-fault-tolerant spanners, dynamic approximate nearest neighbors, dynamic approximate MST;
 88 these algorithms all have $O(\log n)$ time per update. For several of these problems, they were the first to
 89 achieve logarithmic update time.

90 The existence of an LSO in Euclidean metrics naturally motivates the question of constructing an LSO
 91 for doubling metrics. While doubling metrics vastly generalize Euclidean metrics, many nice properties
 92 of Euclidean geometry are lost in doubling metrics. The technique of Chan, Har-Peled and Jones [10]
 93 relied extensively on Euclidean geometry, and it is unclear if their technique can be easily extended to
 94 doubling metrics. Nevertheless, they are able to construct a (τ, ε) -LSO for point sets in doubling metrics
 95 of dimension λ where the number of orderings is $\tau = O(\log(n)/\varepsilon)^{O(\lambda)}$, which depends on the number
 96 of points. An open problem left by their work is to reduce the number of orderings to¹ $O_{\varepsilon, \lambda}(1)$. This
 97 problem was recently solved by Filtser and Le [16]; the number of orderings in their construction is
 98 $\tau = \varepsilon^{-O(\lambda)}$ orderings. Their LSO (and its variants) are powerful primitives to solve various problems in
 99 metric spaces [16, 15].

100 An arguably more important problem is to construct a *dynamic LSO* with a small number of orderings
 101 in doubling metrics. As mentioned above, a dynamic LSO will give dynamic algorithms for a host of
 102 problems in doubling metrics, recovering the full power of LSO. If all possible points in the metric occurring

¹We use the notation $O_{\varepsilon, \lambda}$ to hide the dependency on ε and λ .

103 during the course of the algorithm are given in advance, that is, the points under insertions/deletions
 104 belong to a specific set P given at the beginning of the algorithm, Chan, Har-Peled, and Jones [10] gave
 105 a simple dynamic LSO by simulating their Euclidean counterpart. The key observation is that if P is
 106 given, one can construct a net tree, which then can be used to “partition the space” in the same way that
 107 a quadtree partitions \mathbb{R}^d . However, knowing P in advance is a very strong and artificial assumption for a
 108 dynamic data structure, as concurred by Chan, Har-Peled, and Jones [10].

109 Another major problem of the dynamic LSO for doubling metrics by Chan, Har-Peled, and Jones [10]
 110 is the size: the number of orderings is poly-logarithmic instead of a constant. On the other hand, for the
 111 LSO of constant size by Filtser and Le [16], achieving a *static construction* in $O(n \log n)$ time remains an
 112 open problem; their LSO construction is rather complicated, relying on sophisticated objects, such as
 113 ultrametric covers and pairwise partition covers. Therefore, even if all the points are given in advance, it
 114 is not easy to dynamize the construction of Filtser and Le with a poly-logarithmic time per update. We
 115 note that it might be possible to construct the LSO by Filtser and Le [16] statically in time $O(n \log(\Delta))$
 116 where Δ is the *spread*² of the point set. However, Δ could be exponential in n , and hence the worst
 117 case running time remains $\Omega(n^2)$. Removing the dependency on Δ is a central problem in designing
 118 algorithms, both static and dynamic, for doubling metrics [20, 12, 18, 19].

119 In this work, we give the first data structure, as formally defined in Definition 2 below, for maintaining
 120 a *dynamic LSO with a constant number of orderings* in doubling metrics. Our data structure could handle
 121 point insertions/deletions to the LSO in $O(\log n)$ time per update.

122 **Definition 2 (Dynamic LSO Data Structure).** (τ, ε) -Dynamic LSO is a data structure maintaining a
 123 (τ, ε) -LSO Σ for a dynamic set of points S and supporting the following operations:

- 124 • $\text{INSERT}(q, \Sigma)$: insert a point q to Σ .
- 125 • $\text{DELETE}(q, \Sigma)$: remove q from Σ .
- 126 • $\text{GETPREDECESSOR}(q, i, \Sigma)$: return the predecessor of q in i^{th} ordering of Σ , return null if q is the
 127 first point in the ordering.
- 128 • $\text{GETSUCCESSOR}(q, i, \Sigma)$: return the successor of q in i^{th} ordering of Σ , return null if q is the last
 129 point in the ordering.

130 One important property of our dynamic LSO is stability. We say that a data structure for maintaining
 131 an LSO of a dynamic point set S is *stable* if for every dynamic ordering σ in the LSO, when a point is
 132 inserted or deleted from S , the data structure does not change the relative ordering of existing points
 133 in the LSO. Intuitively, when a point is deleted from or inserted to S , a stable data structure simply
 134 deleting or inserting it, respectively, from each ordering in the LSO without mixing up the order of other
 135 points. A prior, it is unclear (even in the static setting) a stable LSO exists, and specifically, if there is a
 136 way to insert a new point to an existing LSO to get a new LSO that is also good for the new point. In
 137 some applications of LSO (to be discussed in more detail in Section 1.2) such as dynamic closest pair or
 138 approximate bichromatic closest pair, we consider adjacent pairs of points in all orderings. The stability
 139 of LSO allows us to keep track of these pairs in $O(1)$ time per ordering. On the other hand, without the
 140 stability, we have to update the set of adjacent pairs of points in all orderings, which could cost $\Omega(n)$
 141 time. The stability is even more crucial in dynamic vertex-fault-tolerant (VFT) spanners, since in this
 142 application, we need to query k nearest predecessors and k nearest successors of every point in each
 143 ordering. In the following theorem, which is our main result, we construct a stable dynamic LSO.

²The spread is the ratio between the maximum pairwise distance over the minimum pairwise distance.

144 **Theorem 1.** Given $\varepsilon \in (0, 1)$, there is a data structure maintaining $(\varepsilon^{-O(\lambda)}, \varepsilon)$ -LSO for a dynamic point set
 145 S in doubling metrics of dimension λ supporting INSERT/DELETE in $O(\varepsilon^{-O(\lambda)} \log(n))$ time per operation
 146 and GETPREDECESSOR/GETSUCCESSOR in $O(1)$ time per operation. Furthermore, our LSO is stable.

147 We emphasize that the running time of each operation GETPREDECESSOR/GETSUCCESSOR does not
 148 depend on n, ε or λ . As we will see, in some applications, such as fault-tolerant spanner, achieving
 149 $O(1)$ time per GETPREDECESSOR/GETSUCCESSOR operation as in Theorem 1 is important to get optimal
 150 running time, matching the best static algorithms.

151 **Model assumptions.** Our dynamic algorithm makes the same two common assumptions either explicitly
 152 or implicitly used in prior works in doubling metrics [12, 18, 19]. First, we have access to an exact
 153 distance oracle that, given any two points in the metric, computes their distance in $O(1)$ time. Second,
 154 after a point is deleted from the point set, the distance between the current point and the deleted point
 155 could still be computed in $O(1)$ time. These assumptions can be naturally realized in some special cases
 156 such as low dimensional Euclidean or ℓ_p spaces for a constant $p \geq 1$.

157 Next, we give an overview of our technical ideas for maintaining a dynamic LSO of constant size.
 158 Then in Section 1.2, we discuss the applications of our dynamic LSO; some of these applications were
 159 studied in Euclidean spaces by Chan, Har-Peled and Jones [10].

160 1.1 Key Technical Ideas

161 Our first step is to interpret the (only) existing construction of LSO in doubling metrics by Filtser and
 162 Le [16] in terms of trees since (dynamic) trees are the basic building block of many dynamic algorithms.
 163 Filtser and Le [16] constructed their LSO via a so-called *pairwise partition cover*, which is a family of
 164 hierarchical partitions of the input metric space, and an *ultrametric cover*, which is a family of ultrametrics
 165 that have a certain distance covering property. Their overall construction is rather involved, and it is not
 166 clear even how to implement it statically in $O(n \log n)$ time for constants ε and λ . Here we introduce a
 167 new type of trees and tree covers called *pairwise index tree* (PIT) and *pairwise tree cover*, respectively. A
 168 pairwise tree cover consists of $O(\log(1/\varepsilon))$ different PITs where each pair of points is “covered” by one of
 169 the PITs. In a PIT, each internal node is labeled with one or two points in S , and each leaf is labeled by
 170 exactly one point (to form a bijection to S). For a given pair of points (x, y) , loosely speaking, we would
 171 like to have a node η_{xy} in some PIT T labeled with both x and y such that (the points associated with)
 172 leaves of the subtree rooted at η_{xy} of T is either in the distance $\varepsilon d_X(x, y)$ from x or from y . (Both x
 173 and y will be associated with leaves in the subtree of T rooted at η_{xy} .) If so, then visiting each PIT in
 174 the cover by depth-first search (DFS) would give us a linear ordering of leaves, called *DFS leaf ordering*,
 175 satisfying the locality property (in Definition 1) for x and y : every point between x and y in the DFS
 176 linear ordering will be children of η_{xy} and hence within $\varepsilon d_X(x, y)$. Thus, all the DFS leaf orderings from
 177 all the trees in the pairwise tree cover together would be an LSO. At a more technical level, having such
 178 node η_{xy} for every pair (x, y) would mean the total number of nodes would be $\Omega(n^2)$, rendering any
 179 hope for efficient dynamic maintenance. So we relax it slightly: η_{xy} would be labelled by a pair (x', y')
 180 such that $d_X(x, x'), d_X(y, y') \leq \varepsilon d_X(x, y)$. Both x and y remain associated with leaves of the subtree
 181 rooted at η_{xy} . This relaxation allows many pairs to share the same node and hence could potentially be
 182 maintained efficiently. The formal definitions, therefore, are less intuitive than described here.

183 **Definition 3 (Pairwise index tree (PIT)).** Let $\delta \geq 1$ and ε be parameters. A (δ, ε) -pairwise index tree
 184 of S is a rooted tree with the following properties:

- 185 1. [Pairwise labelling.] *Each node is labeled with one or two elements in S . A node at level i is*
 186 *denoted as (x, y, i) where $x, y \in S$, x could be the same or different from y . A leaf is labeled with*
 187 *exactly one point in S .*
- 188 2. [Packing.] *For two nodes (x, y, i) and (u, v, i) , the distance between any pair of points in $\{x, y, u, v\}$*
 189 *is $\Omega(\frac{\delta}{\varepsilon^{i-1}})$.*
- 190 3. [Covering.] *Label points in all children of (x, y, i) are within the distance $O(\frac{\delta}{\varepsilon^{i-1}})$ from x or y . Let*
 191 *$C_i(x, y)$ be the union of all labels (or leaf labels) in the subtree rooted as (x, y, i) . The diameter of*
 192 *$C_i(x, y)$ is $O(\delta/\varepsilon^i)$. We call $C_i(x, y)$ **the cluster of node (x, y, i)** .*

193 The packing and covering properties in the definition of PIT are very similar to the packing/covering
 194 properties of a net tree, a standard tool for navigating doubling metrics. The key difference between a
 195 PIT and a net tree is that these properties applied to pairs of points. In the construction of pairwise tree
 196 cover, different PITs in the cover will be obtained by varying the parameter δ in Definition 3.

197 Given a PIT T , we say that a node at level i of T , denoted by (u, v, i) , is **ε -close** to a pair (x, y)
 198 if every point $p \in C_i(u, v)$, the cluster of (u, v, i) , has (i) $x, y \in C_i(u, v)$, (ii) $d_X(p, x) \leq \varepsilon d_X(x, y)$ or
 199 $d_X(p, y) \leq \varepsilon d_X(x, y)$ (that is, either p is close to x or close to y).

200 **Definition 4 (Pairwise tree cover).** A (τ, ε) -pairwise tree cover of a point set S , denoted by \mathcal{T} , is a
 201 collection of (δ, ε) -PITs (for different values of δ) such that:

- 202 • [Size.] \mathcal{T} contains at most τ PITs.
- 203 • [Pairwise covering.] For any pair of points $x, y \in S$ whose distance in $[\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i})$ for some $\delta \in$
 204 $\{1, 2^1, 2^2, \dots, 2^{\lceil \log(1/\varepsilon) \rceil}\}$, there exists a (δ, ε) -PIT $T \in \mathcal{T}$ such that a node at level i of T is $O(\varepsilon)$ -
 205 close to pair (x, y) .

206 We remark that there could be more than one (δ, ε) -PITs in a pairwise tree cover \mathcal{T} that shares the
 207 same value of parameter δ ; they are different in internal representations as they cover different sets of
 208 pairs. The key points are (a) there are only $O(\log(1/\varepsilon))$ different values of δ , and (b) as we will show
 209 later, for each δ , there are only $\varepsilon^{-O(\lambda)}$ different PITs sharing the same δ .

210 The pairwise tree cover is our attempt to combine the strengths of the LSO construction by Chan,
 211 Har-Peled and Jones [10] for Euclidean spaces and the LSO construction by Filtser and Le [16] for
 212 doubling metrics. Specifically, Chan, Har-Peled, and Jones [10] constructed a collection of shifted
 213 quadtrees, a well-studied space partitioning data structure in Euclidean spaces, and visited each quadtree
 214 by **Z-order** to form an LSO. The geometrical nature of the quadtree makes it easy to dynamize their static
 215 LSO. An analogous but less powerful counterpart of quadtree in doubling metrics is the net tree. However,
 216 it is unclear if an analogous Z-order in doubling metrics exists. For this reason, Filtser and Le [16]
 217 developed a very different technique to construct an LSO in doubling metrics. First, they constructed a
 218 *pairwise partition cover* that has a certain pairwise property. They then used the partition to construct an
 219 ultrametric covers, and each LSO is constructed from an ultrametric in the cover by induction. Here
 220 we combine the strengths of both works in the PITs: we start with a net tree and augment it with the
 221 pairwise property by Filtser and Le [16], which can be seen as a replacement for the Z-order. Given the
 222 pairwise tree cover, we simply apply a DFS leaf ordering to each tree to get an LSO. Figure 1 (the top
 223 part) illustrates building blocks to construct a (static) LSO for a point set S .

224 In the dynamic setting, we could use the algorithm by Cole and Gottlieb [12] to maintain a dynamic
 225 net tree. As PITs are built on top of net trees, in principle, one could adapt their technique to maintain
 226 a dynamic PIT. In our (static) construction, we show that our PIT has a certain locality condition, and
 227 specifically, the neighborhood of a node in a PIT is a subset of the neighborhood of the corresponding node

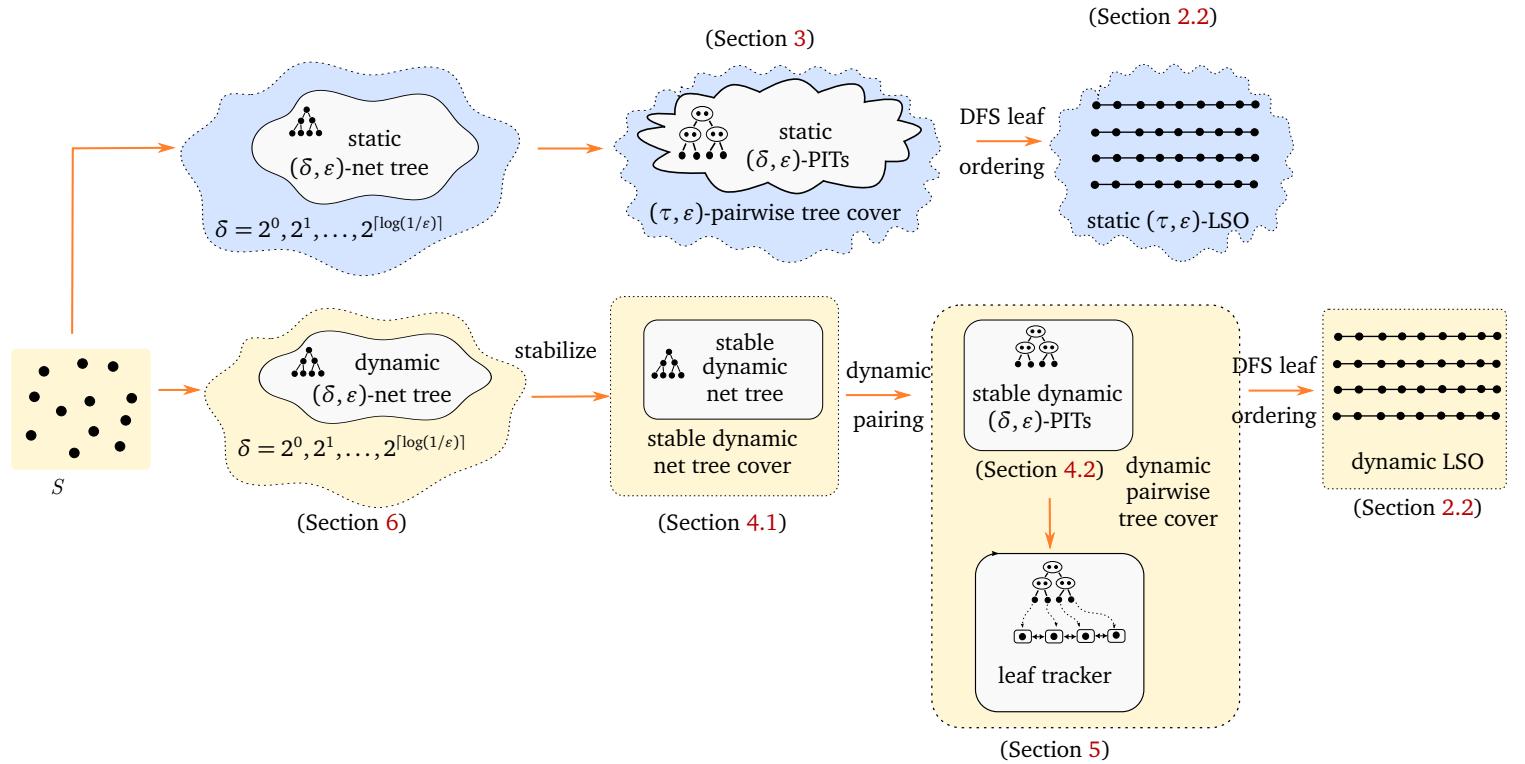


Figure 1: Data structures highlighted in light blue are static while those highlighted in light yellow are dynamic. Data structures with rectangular shapes are stable; others are unstable.

228 in a net tree T . This allows us to maintain a dynamic PIT from a dynamic net tree in a black-box manner.
 229 While the high-level ideas are relatively simple, there are some conceptual difficulties in translating
 230 the static construction to the dynamic construction, mostly due to that a dynamic net tree has to be
 231 compressed and hence some nodes are not directly accessible.

232 The much more difficult task is to maintain a dynamic leaf ordering of PIT, and indeed, all technical
 233 ideas we develop herein are to solve this task. In the static setting, one simply applies DFS to visit
 234 each PIT to get an ordering of the leaves. In the dynamic setting, there are two major challenges (see
 235 Figure 2):

- 236 • (C1): In dynamic net trees (and hence dynamic PITs derived from dynamic net trees), *nodes are*
 237 *only marked deleted* rather than being explicitly deleted from the trees, and hence some leaves
 238 become inactive³ when their corresponding points are deleted from S . In the DFS leaf ordering
 239 of a PIT, we only keep track of an ordering of active leaves. Furthermore, adding a single active
 240 leaf to a net tree could activate $\Omega(n)$ ancestors of the leaf to become active. (A node in the tree is
 241 active if it has at least one active descendant leaf.)
- 242 • (C2): Active descendant leaves of each internal node in a PIT induce a contiguous subsequence of
 243 the DFS leaf ordering of the PIT. When a node x changes parents from u to v as in Figure 2, if we

³Here an inactive leaf refers to a leaf that is marked deleted. Later, in technical sections, sometimes it is convenient for us to insert a null leaf, which is a leaf associated with no point, into a tree. There, an inactive leaf refers to a leaf that is either null or marked deleted.

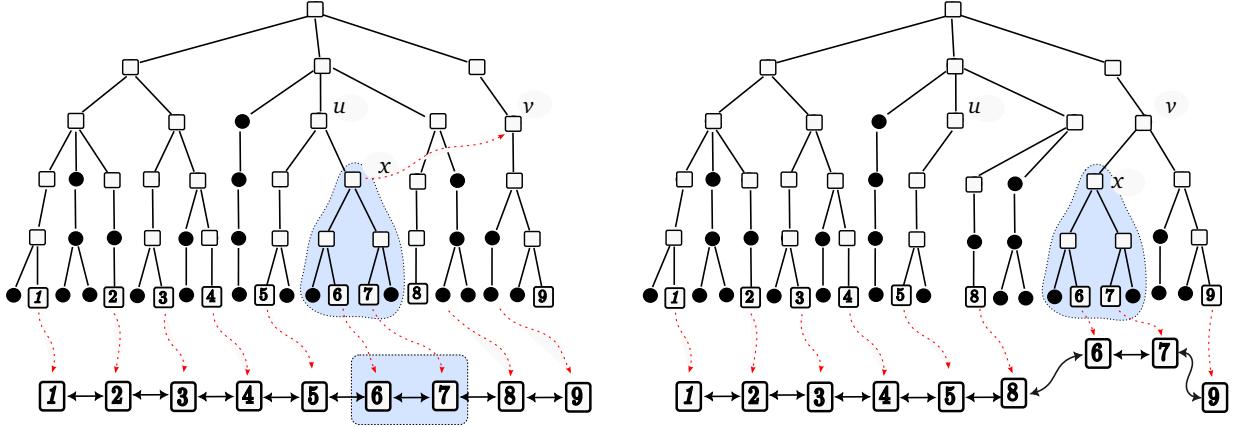


Figure 2: A node x in a PIT changes parent from u to v leads to changes in DFS leaf orderings of v and its ancestors, and of u and its ancestors. Rectangular nodes are active nodes, which either leaves corresponding to non-deleted points or internal nodes with at least one active descendant leaf.

can identify the leftmost active leaf, say l_v , of v in its subsequence, then we could simply slice the DFS leaf subsequence of x and stitch it to the left of l_v in the DFS leaf ordering. (This means that the recourse of a parent update in the DFS leaf ordering is small.) The difficulty is in identifying the leftmost (and also rightmost) leaf of v , and more generally, of an internal node. A natural idea is to a pointer from each internal node to the leftmost/rightmost descendant leaves, but *a parent update of a single node in a PIT could change the pointers to the leftmost/rightmost descendant leaves of all of its (both old and new) ancestors*. There could be $\Omega(n)$ such ancestors (if the aspect ratio Δ is large).

In the special case of incremental dynamic, a.k.a. insertion only, (C1) does not happen since there is no deletion, it remains challenging to resolve (C2). A well-known technique for maintaining a certain kind of DFS of a dynamic tree is the Euler tour technique [35, 21]. However, the Euler tour maintains a DFS ordering of all edges in the tree (in both directions), while we only maintain a list of leave nodes, and hence we need to be able to query a leftmost leaf in the subtree of an internal node. There seems to be no easy way to modify existing dynamic tree data structures, including the Euler tour technique, for this purpose under parent updates. This problem is significantly compounded by the presence of deleted leaves: even designing a data structure for querying an (arbitrary) active descendant leaf of a given internal node in a PIT becomes non-trivial.

We will take several steps to resolve both challenges, see the lower part of Figure 1. Our key idea is to stabilize the net tree using a net tree cover. The formal definition of a net tree cover is somewhat unintuitive and hence we defer it to Definition 8 in Section 4; here we describe its high-level intuition. A dynamic net tree cover is a collection of $O_\lambda(1)$ net tree where updates to every tree are restricted to one of three types: inserting a new leaf, marking a leaf deleted, and subdividing an edge. Therefore, the only non-trivial parent update in the tree is by edge subdivision: inserting a new node z in the middle of an edge (x, y) between a parent x and a child y , effectively changing the parent of y from x to z . This type of parent change does not alter the DFS leaf ordering of a node in the tree, which is the key to our dynamic data structure. We say that every tree in the net tree cover is *stable*; updates that are not one of the three types above are *unstable*. As stable updates are too restrictive, it should not be surprising that the dynamic net tree by Cole and Gottlieb [12], as well as many other dynamic tree data structures, are unstable.

Our basic idea is to “decompose” an (unstable) update to a net tree T into $O_\lambda(1)$ stable updates to \mathcal{J} ,

the corresponding net tree cover. The main observation is that when a node $(p, i-1)$ at level $i-1$ of a net tree T changes its parent at level i from (u, i) to (v, i) , then both $d_X(p, u)$ and $d_X(p, v)$ are bounded by $O(\delta/\varepsilon^i)$ and hence small compared to the radius at level i , by the covering of the net tree. Suppose in an *ideal situation* in which we have a version of T , denoted by J , where we only keep a $\frac{c\delta}{\varepsilon^i}$ -net at level i of J for a sufficiently big constant c , and that u happens to be in the net, then both p and v will be children of u , and there is no need to change the parent of $(p, i-1)$ to (v, i) . (The bound $\frac{\delta}{\varepsilon^i}$ is the packing/covering radius at level i of a (δ, ε) -net tree; see Definition 6.) Of course, the ideal situation will not always happen, and therefore, we construct many, but $O_\lambda(1)$, different versions of T in the net tree cover, and we could show that, loosely speaking, the ideal situation will happen at one version of T . Our idea is to realize some kind of shifting strategy, in the same way the shifted quadtree was used in Euclidean space by Chan, Har-Peled, and Jones [10]. However, we do not have Euclidean geometry; instead, we use a standard coloring trick (e.g., [3, 24]) to color net points. To implement all of these ideas, we have to handle two major difficulties: (1) the ideal situation only happens at one of the trees, and we have to handle non-ideal situations in other trees—the key to this is that we have more leeway in other trees, as the important pairs were taken care of in the ideal situation; (2) when new node arrives due to insertions of new points to S , one has to *merge* it with other nodes, leading to unstable parent changes. (Roughly speaking, a node (v, i) is merged to a node (u, i) if (v, i) 's children become (u, i) 's children in a version J of T .) We resolve both problems by developing two rules on top of the net tree cover, namely *merging by distances* and *merging through time*: when a new node arrives, we look at its distance to existing nodes, and decide to merge based on both the distances and the time when the node arrives. All in all, we are able to show that updates to J are stable. As stability is the key to dynamic algorithms, we believe that this construction is of independent interest.

Given a stable dynamic net tree (encapsulated in a dynamic net tree cover), we develop a dynamic algorithm, called *dynamic pairing*, to construct dynamic PITs. Here, we exploit the stability of the net tree cover to simplify and adapt our static construction to the dynamic setting. As we noted earlier, our static construction has a certain locality condition. An important guarantee of our dynamic pairing algorithm is that (updates to) our dynamic PITs are stable, given that the input dynamic net tree is stable; this is important for the next step: keeping track of DFS ordering of active leaves in PITs.

Finally, we develop a data structure, called *leaf tracker*, to keep track of the DFS ordering of active leaves in a PIT. Recall that the ordering is obtained by visiting each PIT by DFS, breaking ties by the insertion time. We will store the DFS ordering using a *doubly linked list* σ . We also build a skip list on top of σ to perform some kind of binary search. We design the keys to the skip lists to be what we call *ancestral arrays*. Roughly speaking, an *ancestral array* of a node $u \in T$ is an array $O(\log n)$ “important” ancestors stemming from a centroid decomposition of T (see Definition 11). Though there is no linear order between the ancestral arrays to use them as keys in the traditional sense, we could use them to determine if a leaf x is a descendant of a query node u or not by Lemma 14, which turns out to be sufficient for binary search using skip lists. There are several subtleties in the implementation, which we will discuss in detail later in Section 5.1. Here the stability of the updates in PITs helps in two ways: (i) only leaves get inserted into a PIT and hence the ancestral arrays of a node do not change by much after an insertion; and (ii) a node could only change parent due to edge subdivision, but edge subdivision does not change the DFS leaf ordering σ . Therefore, we could rely on the data structure of Kopelowitz and Lewenstein [25] to maintain ancestral arrays (and their associated centroid decomposition) under stable updates in $O(\log n)$ time.

Now, to keep track of the DFS leaf ordering, when a new node q is inserted to a PIT as a leaf child of a node u , we will locate the current rightmost active descendant leaf, say r_u of u in σ —assume for now that u has at least one such leaf—and then insert q after r_u in σ . The basic idea is to first find an arbitrary active descendant leaf x of u and then start a binary search procedure to search for r_u in the skip list using ancestral arrays as keys. In the case where the parent u of q does not have any active

322 descendant leaf before inserting q , our idea is to search for the lowest ancestor, say v , of u that has at
 323 least one active descendant leaf, find the leftmost/rightmost active descendant leaf of v , and insert q next
 324 to the leftmost/rightmost leaf. Finding an arbitrary active descendant leaf and the lowest active ancestor
 325 are rather non-trivial: in the former case, u might contain up to $\Omega(n)$ inactive descendant leaves, while
 326 in the latter case, we might end up checking a large number of ancestors of u . Here we develop a new
 327 data structure called *active tracker* to support both operations.

328 With all ideas together, we are able to develop a dynamic data structure for a pairwise tree cover,
 329 formally defined in Definition 5 below, which maintains a set of stable dynamic PITs and their corre-
 330 sponding DFS leaf orderings (using leaf trackers). One important corollary is that the DFS leaf ordering
 331 of every stable PIT is also stable: we say that a *DFS leaf ordering is stable* if the insertion or deletion of a
 332 new active leaf does not change the relative DFS ordering of all existing active leaves. The stability of
 333 the DFS leaf ordering is because edge subdivision does not change DFS leaf ordering and inserting a new
 334 active leaf does not change the relative order of existing nodes. See Figure 1 for a graphical illustration
 335 of all ideas.

336 **Definition 5 (Dynamic pairwise tree cover data structure).** *A data structure for maintaining a (τ, ε) -*
 337 *pairwise tree cover \mathcal{T} and a *stable DFS leaf ordering* of every PIT in \mathcal{T} , and supporting the following*
 338 *operations:*

- 339 • $\text{INSERT}(q, \mathcal{T})$: *insert a new point q to \mathcal{T} .*
- 340 • $\text{DELETE}(q, \mathcal{T})$: *remove an existing point q from \mathcal{T} .*
- 341 • $\text{GETPREDECESSOR}(q, i, \mathcal{T})$: *return the predecessor of q in σ_i where σ_i is the DFS leaf ordering of*
 342 *the i^{th} tree of \mathcal{T} . The result is null if q is the first element in σ_i .*
- 343 • $\text{GETSUCCESSOR}(q, i, \mathcal{T})$: *return successor of q in σ_i where σ_i is the DFS leaf ordering of the i^{th}*
 344 *tree of \mathcal{T} . The result is null if q is the last element in σ_i .*

345 Since we maintain DFS leaf orderings by a doubly linked list, we can support querying the predecessor
 346 or successor of a point in $O(1)$ time. The following theorem, whose proof will be given in Section 4,
 347 summarizes our main technical result.

348 **Theorem 2.** *Given $\varepsilon > 0$, there is a data structure maintaining (τ, ε) -pairwise tree cover with $\tau = \varepsilon^{-O(\lambda)}$*
 349 *supporting $\text{INSERT}/\text{DELETE}$ in $O(\varepsilon^{-O(\lambda)} \log(n))$ time per operation, and $\text{GETPREDECESSOR}/\text{GETSUCCESSOR}$*
 350 *in $O(1)$ time per operation.*

351 As we mentioned above, each ordering in an LSO is basically a DFS ordering of (active) leaves in a PIT.
 352 Hence, once we can maintain PITs and their DFS leaf orderings, we could obtain an LSO as a corollary.
 353 As the DFS leaf orderings are stable, the LSO we obtain is also stable. The following theorem formalizes
 354 our results; the proof is rather simple, and will be given in the preliminaries section (Section 2.2).

355 **Theorem 3.** *If there is a data structure for dynamic (τ, ε) -pairwise tree cover supporting $\text{INSERT}/\text{DELETE}$ in $T_1(n, \varepsilon)$ time per operation and $\text{GETPREDECESSOR}/\text{GETSUCCESSOR}$ in $T_2(n, \varepsilon)$ time per operation, then*
 356 *we can construct a data structure for $(\tau, O(\varepsilon))$ -LSO supporting $\text{INSERT}/\text{DELETE}$ in $O(T_1(n, O(\varepsilon)))$ time*
 357 *per operation, and $\text{GETPREDECESSOR}/\text{GETSUCCESSOR}$ in $O(T_2(n, O(\varepsilon)))$ time per operation. Furthermore,*
 358 *the LSO is stable (due to the stability of the DFS leaf orderings in the pairwise tree cover).*

360 We observe that Theorem 1 follows directly from Theorem 3 and Theorem 2.

361 1.2 Applications

362 We now give examples of applications of our dynamic LSO in Theorem 1. We note that the list of
 363 applications mentioned here is not meant to be exhaustive. We believe that LSO could find many more
 364 applications in handling dynamic point sets in doubling metrics. As we remarked earlier, the stability of
 365 our dynamic LSO is the key to applications. All but the dynamic tree cover application were shown for
 366 Euclidean spaces by Chan, Har-Peled and Jones [10].

367 **Dynamic vertex-fault-tolerant spanners.** Given a set of points S in a doubling metric of dimension λ ,
 368 we denote by G_S the complete graph representing the submetric induced on S . A t -spanner of S is a
 369 spanning subgraph H of G_S such that $d_H(x, y) \leq t \cdot d_{G_S}(x, y)$ for every $x, y \in S$. Given $k \in [1, n-2]$,
 370 we say that H is an *k-fault-tolerant t-spanner*, or (k, t) -VFTS for short, if for every subset $F \subset S$ of size at
 371 most k , called a *faulty set*, $H \setminus F$, the graph obtained by removing every vertex in F from H , is a $(1 + \varepsilon)$
 372 of $S \setminus F$.

373 Observe that in a $(k, 1 + \varepsilon)$ -VFTS, every vertex must have a degree at least k , and therefore at least
 374 $\Omega(nk)$ edges. Levcopoulos, Narasimhan, and Smid [28] introduced and constructed the first k -fault-
 375 tolerant $(1 + \varepsilon)$ -spanner in Euclidean spaces of constant dimensions that has $O(k^2n)$ edges. There was
 376 then a long line of work, see e.g., [29, 13, 8, 9, 34, 27], aiming to improve the Euclidean construction by
 377 Levcopoulos, Narasimhan, and Smid as well as extend their result to doubling metrics. Specifically, in
 378 doubling metrics, it is possible to achieve degree bound $O(k)$ and/or $O(nk)$ number of edges [34, 27].
 379 Some constructions are *simple* but could only achieve $O(nk)$ number of edges (without any reasonable
 380 bound on the degree) [8], or the degree is $\Omega(k^2)$ [9]; other constructions achieving optimal degree
 381 bound of $O(k)$ (for constant ε and λ) are sophisticated [34].

382 A more ambitious goal is to construct an optimal k -fault-tolerant $(1 + \varepsilon)$ -spanner efficiently, even in
 383 the static setting. Solomon [34] devised an $O(n(\log n + k))$ -time algorithm to construct a k -fault-tolerant
 384 $(1 + \varepsilon)$ -spanner with degree $O(k)$ (and diameter $O(\log k)$ and lightness $O(k^2 \log n)$); the running time is
 385 of Solomon's algorithm is optimal in both n and k . Solomon's result settled an important open problem
 386 raised in the book of Narasimhan and Smid (Problems 26 and 27 in [31]). Recently, Le, Solomon, and
 387 Than [27] designed a different algorithm with the same running time but achieving both optimal degree
 388 and lightness. Given the slow progress on static algorithms, it is understandable that the problem of
 389 maintaining a dynamic (k, t) -VFTS in doubling metrics under point updates remains wide open. Even
 390 maintaining a dynamic and *non-fault-tolerant* spanner, a much simpler problem, proved to be very
 391 challenging. Gottlieb and Roddity [19] were the first to achieve $O(\log n)$ time per update after several
 392 attempts [32, 18, 19]. Their dynamic algorithm is much more complicated than its static counterpart [7].
 393 It is, therefore, unlikely that their technique could be extended to handle (k, t) -VFTS.

394 Given our dynamic LSO in Theorem 1 as a black box, following [10], we obtain a dynamic algorithm
 395 for maintaining $(k, 1 + \varepsilon)$ -VFTS in $O_{\lambda, \varepsilon}(\log n + k)$ time per update in a very simple way: for each point
 396 $p \in S$, add edges to its $k + 1$ predecessors and $k + 1$ successors in each ordering of Σ . As $|\Sigma| = \varepsilon^{-O(\lambda)}$,
 397 our dynamic $(k, 1 + \varepsilon)$ -VFTS spanners achieve both *optimal running time per update*, *optimal degree* (and
 398 hence the number of edges), and *optimal running time to query all neighbors of a vertex*.

399 **Theorem 4.** *Given $\varepsilon \in (0, 1)$, $k \in [1, n-2]$ and a dynamic point set S in doubling metrics of dimension
 400 λ , there is a data structure \mathcal{D} such that \mathcal{D} (implicitly) maintains a $(k, 1 + \varepsilon)$ -VFTS H of degree $k \cdot \varepsilon^{-O(\lambda)}$
 401 for S in $O(\log n \varepsilon^{-O(\lambda)})$ time per update, and \mathcal{D} returns all neighbours of a given vertex of H in $k \varepsilon^{-O(\lambda)}$
 402 time. The update time and query time are optimal for fixed ε, λ .*

403 **Dynamic tree covers.** This result is an application of our technique rather than a direct application
 404 of LSO. Given a set of points S in a doubling metric (X, d_X) , a *tree cover* for S is a collection of edge-

405 weighted trees \mathcal{T} such that for every tree $T \in \mathcal{T}$, $S \subseteq V(T)$ and $d_X(x, y) \geq d_T(x, y)$. The *size* of the tree
 406 cover \mathcal{T} , denoted by $|\mathcal{T}|$, is the number of trees in \mathcal{T} . The *stretch* of \mathcal{T} is the smallest $t \geq 1$ such that
 407 $d_X(x, y) \leq t \cdot \min_{T \in \mathcal{T}} d_T(x, y)$. Tree covers have been studied extensively both in general metrics [36]
 408 and special metrics, such as Euclidean [1], planar [3, 11], and doubling [3, 24]. Tree covers also have
 409 many algorithmic applications, such as spanners, routing, and distance oracles; see [24] for a thorough
 410 discussion. In doubling metrics, a tree cover for n points could be (statically) constructed in $O(n \log n)$
 411 time [24]. However, there is no known dynamic construction of tree covers. Indeed, dynamically
 412 maintaining a tree cover is at least as hard as maintaining a dynamic spanner, which, as discussed above,
 413 is a difficult problem. Our technique for maintaining a dynamic *pairwise tree cover* could be adapted
 414 directly to maintain a tree cover with $O_{\lambda, \varepsilon}(\log n)$ per point update.

415 **Theorem 5.** *Given a dynamic point set S in doubling metrics of dimension λ and any $\varepsilon \in (0, 1)$, there is
 416 a data structure \mathcal{D}_J explicitly maintaining a tree cover J for S such that J has stretch of $1 + \varepsilon$ and size of
 417 $\varepsilon^{-O(\lambda)}$, and the running time per update is $O(\varepsilon^{-O(\lambda)} \log(n))$.*

418 **Closest pair of points.** Finding the closest pair in a point set is a very well-studied problem in
 419 computational geometry. In Euclidean spaces, there is a vast amount of literature on this problem. In
 420 the Euclidean spaces of constant dimension, the dynamic closest pair can be maintained in $O(\log n)$ time
 421 per update [4, 17, 10]. In metrics of bounded doubling dimension, there are two fast static algorithms
 422 for finding the closest pair: one based on well-separated pair decomposition (WSPD) [20], and the
 423 other is divide and conquer [33]. Both algorithms are *randomized* and have an expected running time
 424 of $O(n \log n)$. Using our dynamic LSO, we could maintain the closest pair in $O(\log n)$ time per update.
 425 Applying our dynamic algorithm to the static setting, we obtain a *deterministic* algorithm for the closest
 426 pair in metrics of bounded doubling dimension in time $O_{\lambda, \varepsilon}(n \log n)$.

427 **Theorem 6.** *Given a dynamic point set S in doubling metrics of dimension λ , we construct a data
 428 structure for maintaining the closest pair in S in $2^{O(\lambda)} \log(n)$ time per update.*

429 **Bichromatic closest pair of points.** This is another fundamental problem in computational geometry:
 430 given two point sets R (red) and B (blue) in a metric space, find the closest pair of points, one red and
 431 one blue, among all red-blue pairs of points. In Euclidean metrics, both static and dynamic versions of
 432 this problem have been studied extensively (see, e.g. [14] and references therein). However, in doubling
 433 metrics, there is no known dynamic algorithm for this problem. Here we use our dynamic LSO to provide
 434 the first approximate dynamic algorithm.

435 **Theorem 7.** *Given a parameter $\varepsilon \in (0, 1)$ and two dynamic point sets R, B in doubling metric of
 436 dimension λ , there is a data structure such that it maintains $(1 + \varepsilon)$ -closest pair (r, b) where $r \in R, b \in B$,
 437 and runs in $O(\varepsilon^{-O(\lambda)} \log(n))$ per update of R or B , where $n = |R| + |B|$.*

438 **Approximate nearest neighbor search.** One problem that motivated the early study of dynamic
 439 algorithms for point sets in doubling metrics is the approximate nearest neighbor: given a query point p ,
 440 find a point q such that $d_X(p, q) \leq (1 + \varepsilon) \min_{x \in X \setminus p} d_X(p, x)$. We would like to design a dynamic data
 441 structure that could support fast update time and query time. The pioneering work of Krauthgamer and
 442 Lee [26] proposed the first dynamic solution for this problem with $O_{\varepsilon, \lambda}(\log \Delta \log \log(\Delta))$ update time
 443 and $O(\log \Delta + \varepsilon^{-O(\lambda)})$ query time, which is optimal. Cole and Gottlieb [12] then removed the dependency
 444 on the spread Δ and improved the update time to $O_{\varepsilon, \lambda}(\log n)$ and the query time to $O(\log(n) + \varepsilon^{-O(\lambda)})$.
 445 Ideally, we would like the query time to be much faster than the update time; for example, in database
 446 applications, querying nearest neighbors is done much more frequently than deleting/inserting points.

447 Our dynamic LSO in Theorem 1 gives a simple solution for this problem. For a given query point p , the
 448 idea is to first insert p to the current LSO of the point set, return the closest neighbor in the orderings of
 449 p , and then delete p from the LSO. As the number of orderings is $O_{\varepsilon, \lambda}(1)$, and p has at most 2 neighbors
 450 per ordering, the query time is $O_{\varepsilon, \lambda}(1)$, plus the time to insert and delete p from the LSO, which is
 451 $\varepsilon^{-O(\lambda)} \log(n)$. We note that $\log(n)$ query time is optimal for a constant ε, λ for any data structure with
 452 linear space [2].

453 **Theorem 8.** *Given a dynamic point set S in doubling metrics of dimension λ , we can construct a $(1 + \varepsilon)$ -
 454 nearest neighbor data structure for supporting point deletions/insertions in $O(\varepsilon^{-O(\lambda)} \log(n))$ time per
 455 update, and $\varepsilon^{-O(\lambda)} \log(n)$ query time.*

456 2 Preliminaries

457 2.1 Basic Notation

458 Given a metric space (X, d_X) , let Δ be the ratio between the maximum and the minimum distance in
 459 the space. A ball of p radius r is a set of all points in distance r from p : $B(p, r) = \{q \in X : d(p, q) \leq r\}$.
 460 (X, d_X) has doubling dimension λ if any ball with radius $2r$ can be covered by at most 2^λ balls of radius
 461 r . The *packing property* of a doubling metric states that any set of points with maximum distance R and
 462 minimum distance r has at most $\left(\frac{4R}{r}\right)^\lambda$ points.

463 Y is a *r-net* of point set S if Y is a subset of S such that: (i) for all $x, y \in Y$ and $x \neq y$, $d_X(x, y) > r$
 464 (this property is called *packing*), (ii) for every point $x \in S$, there exists a point $y \in Y$ such that $d_X(x, y) \leq r$
 465 (this property is called *covering*). *Net tree* is a hierarchical tree where: each node has a label where the
 466 set of leaf labels is a bijection into S , and the set of points at level i , denoted as Y_i , is r^i -net of Y_{i-1} . We
 467 denote a node by a pair (t, i) , where t is a point in S and i is the level of the node. Sometimes we simply
 468 use t instead of (t, i) when the level is clear from the context. The distance between two nodes in the
 469 net tree means the distance between two points labeled these nodes.

470 For a dynamic point set S , Cole and Gottlieb [12] showed how to construct a net tree with relaxed
 471 packing and covering properties: (i) for $x, y \in Y_i$ and $x \neq y$, $d_X(x, y) > \alpha \frac{1}{\varepsilon^i}$, (ii) for $x \in Y_{i-1}$, there
 472 exists $y \in Y_i$ such that $d_X(x, y) \leq \phi \frac{1}{\varepsilon^i}$, where α and ϕ are some constants. In this work, we use the
 473 notion of *(δ, ε) -net tree* to mention the net tree with relaxed packing and covering properties, and Y_i is
 474 the $\frac{\delta}{\varepsilon^i}$ -net of Y_{i-1} for any level i .

475 **Definition 6 $((\delta, \varepsilon)$ -net tree).** *The (δ, ε) -net tree is a net tree with packing and covering properties as
 476 follows:*

- 477 • [Packing.] two nodes $(x, i), (y, i)$ have $d_X(x, y) > \Omega(\frac{\delta}{\varepsilon^i})$.
- 478 • [Covering.] if (x, i) is the parent of $(y, i-1)$, then $d_X(x, y) \leq O(\frac{\delta}{\varepsilon^i})$.

479 2.2 Dynamic LSO from Dynamic Pairwise Tree Cover

480 We now show how to construct LSO from a pairwise tree cover; the proof addresses both static and
 481 dynamic settings.

482 **Theorem 3.** *If there is a data structure for dynamic (τ, ε) -pairwise tree cover supporting INSERT/DELETE
 483 in $T_1(n, \varepsilon)$ time per operation and GETPREDECESSOR/GETSUCCESSOR in $T_2(n, \varepsilon)$ time per operation, then
 484 we can construct a data structure for $(\tau, O(\varepsilon))$ -LSO supporting INSERT/DELETE in $O(T_1(n, O(\varepsilon)))$ time
 485 per operation, and GETPREDECESSOR/GETSUCCESSOR in $O(T_2(n, O(\varepsilon)))$ time per operation. Furthermore,
 486 the LSO is stable (due to the stability of the DFS leaf orderings in the pairwise tree cover).*

487 **Proof:** We show how to construct $(\tau, O(\varepsilon))$ -LSO from a (τ, ε) -pairwise tree cover \mathcal{T} , then we obtain a
 488 (τ, ε) -LSO by scaling ε with a constant factor.

489 First, we show a static construction.

490 Given a (τ, ε) -pairwise tree cover $\mathcal{T} = \{T_1, T_2, \dots, T_\tau\}$, let Σ be the set of $\{\sigma_1, \sigma_2, \dots, \sigma_\tau\}$,
 491 where σ_i is the DFS leaf ordering of $T_i \in \mathcal{T}$.

492 By the covering of the pairwise tree cover, for any pair x, y with $d_X(x, y) \in [\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i}]$, there is a tree T_j
 493 such that a node at level i of T_j is $O(\varepsilon)$ -close to (x, y) . Let that node be (u, v, i) . Recall that $C_i(u, v)$
 494 is a set of all leaf labels under the subtree rooted at (u, v, i) and $x, y \in C_i(u, v)$. By DFS, all points
 495 in $C_i(u, v)$ are written consecutively in σ_j . This implies that any point p between x and y in σ_j has
 496 $d_X(p, x) \leq O(\varepsilon)d_X(x, y)$ or $d_X(p, y) \leq O(\varepsilon)d_X(x, y)$. Therefore, Σ is a $(\tau, O(\varepsilon))$ -LSO.

497 Here is the dynamic maintenance for Σ :

498 Suppose that we are given a data structure $\mathcal{D}_\mathcal{T}$ maintaining the (τ, ε) -pairwise tree cover \mathcal{T}
 499 under insertions and deletions. Our data structure \mathcal{D}_Σ maintaining $(\tau, O(\varepsilon))$ -LSO Σ dynamically
 500 invokes operations of $\mathcal{D}_\mathcal{T}$ directly. To update a point p , $\text{INSERT}(p, \Sigma)$ calls $\text{INSERT}(p, \mathcal{T})$,
 501 and $\text{DELETE}(p, \Sigma)$ calls $\text{DELETE}(p, \mathcal{T})$. To access orderings, we use operations of getting
 502 the predecessor or the successor of \mathcal{D}_Σ , where $\text{GETPREDECESSOR}(p, i, \Sigma)$ and $\text{GETSUCCESSOR}(p, i, \Sigma)$
 503 of \mathcal{D}_Σ return the result of $\text{GETPREDECESSOR}(p, i, \mathcal{T})$ and $\text{GETSUCCESSOR}(p, i, \mathcal{T})$
 504 respectively.

505 The running time follows directly from the construction. The stability of the LSO follows from that
 506 of DFS leaf ordering of \mathcal{T} , implying the theorem. \square

507 3 Pairwise Tree Cover: Static Construction

508 In this section, we show the static construction for a collection of (δ, ε) -PITs as claimed in Theorem 9. Then we construct an $(\varepsilon^{O(-\lambda)}, \varepsilon)$ -pairwise tree cover by simply constructing PITs for each
 509 $510 \delta \in \{1, 2^1, 2^2, \dots, 2^{\lceil \lg(1/\varepsilon) \rceil}\}$.

511 **Theorem 9.** *Given a (δ, ε) -net tree T of a point set S , we can construct from T a collection of (δ, ε) -PITs
 512 \mathcal{T} with $\varepsilon^{-O(\lambda)}$ trees such that for any pair of points (x, y) whose distance in $[\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i}]$, there exist a PIT T'
 513 in the collection and a node at level i of T' that is $O(\varepsilon)$ -close to (x, y) .*

514 We call a node in a PIT T' as a *pairwise node*, to distinguish with a node in net tree T . We simply
 515 refer to a pairwise node as a node when the tree in the context is a PIT. A pairwise node at level i can be
 516 labeled by a single point of the form (p, p, i) or two different points of the form (x, y, i) . In the former
 517 case, we say that the node has a *single-label* and in the latter case, *double-label*. For a given point $p \in S$,
 518 we define *the node (pairwise node) at level i of point p* to be the ancestor at level i of the leaf $(p, 0)$
 519 (($p, p, 0$, resp.) in the net tree T (PIT T' , resp.). If (u, v, i) is the pairwise node at level i in T' of p ,
 520 and $p \in Y_{i-1}$, we also say *(u, v, i) is the pairwise node in T' of $(p, i-1)$* (in T), for some $u, v \in Y_{i-1}$; it
 521 could be that $p \notin \{u, v\}$, and hence it is *not* always the case that the corresponding pairwise node of
 522 (p, i) is labeled with the same point p . Observe that level i of a PIT corresponds to level $i-1$ in the
 523 corresponding net tree; they are off by one level.

524 First, we describe intuitively an $O(\varepsilon)$ -close node for a pair of points x_0, y_0 with $d_X(x_0, y_0) \in [\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i}]$.
 525 We denote by Y_i the set of net points at level i ; these are points associated with nodes at level i of T .
 526 Recall that in (δ, ε) -net tree, given a level i , any point p_0 in S has a node (p, i) such that $p_0 \in B(p, \frac{2\delta}{\varepsilon^i})$,
 527 we say the ball of (p, i) *covers* p_0 . Let $(x, i-1)$ and $(y, i-1)$ be nodes at level $i-1$ whose balls cover

528 x_0 and y_0 , respectively. Observe that for every $p \in B(x, \frac{2\delta}{\varepsilon^{i-1}})$, $d_X(x_0, p) \leq \frac{4\delta}{\varepsilon^{i-1}}$ by triangle inequality.
529 Similarly, for every $p \in B(y, \frac{2\delta}{\varepsilon^{i-1}})$, $d_X(y_0, p) \leq \frac{4\delta}{\varepsilon^{i-1}}$. Therefore, if we have a pairwise node in a PIT such
530 that its cluster is the union of $B(x, \frac{2\delta}{\varepsilon^{i-1}})$ and $B(y, \frac{2\delta}{\varepsilon^{i-1}})$, this node is $O(\varepsilon)$ -close to (x_0, y_0) . To see this,
531 suppose that we have a node (x, y, i) in the PIT and its cluster $C_i(x, y) = B(x, \frac{2\delta}{\varepsilon^{i-1}}) \cup B(y, \frac{2\delta}{\varepsilon^{i-1}})$, then it
532 satisfies: **(i)** $x_0, y_0 \in C_i(x, y)$, **(ii)** for any $p \in C_i(x, y)$, $d_X(p, x_0)$ or $d_X(p, y_0)$ is at most $\frac{4\delta}{\varepsilon^{i-1}}$, which is
533 $4\varepsilon^{\frac{\delta}{\varepsilon^i}} = O(\varepsilon)d_X(x_0, y_0)$.

534 Now, we sketch our main idea for the static construction. Observe by the triangle inequality,
535 $\|d_X(x, y) - d_X(x_0, y_0)\| \leq d_X(x, x_0) + d_X(y, y_0)$, which means $d_X(x, y) \in [\frac{\delta}{\varepsilon^i} - \frac{4\delta}{\varepsilon^{i-1}}, \frac{2\delta}{\varepsilon^i} + \frac{4\delta}{\varepsilon^{i-1}}]$. To
536 have $O(\varepsilon)$ -close nodes for all pairs with the distance in $[\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i}]$, we consider all pairs u, v of Y_{i-1} , for
537 each pair if $d_X(u, v) \in [\frac{\delta}{\varepsilon^i} - \frac{4\delta}{\varepsilon^{i-1}}, \frac{2\delta}{\varepsilon^i} + \frac{4\delta}{\varepsilon^{i-1}}]$, then we create a pairwise node (u, v, i) . Next, we arrange
538 these nodes into PITs such that: **(i)** for each PIT, every point in Y_{i-1} belongs to the label of at most one
539 level- i pairwise node, **(ii)** clusters of nodes at level i are the union of some clusters of nodes at level $i-1$,
540 **(iii)** these level- i clusters are disjoint. Then, in each PIT, for any point $p \in Y_{i-1}$ that does not belong to
541 a cluster of any pairwise node at level i , we create a single-label node (p, p, i) in that PIT. Finally, we
542 create edges connecting pairwise nodes at level i and level $i-1$: if the cluster of a node u at level $i-1$
543 is a subset of the cluster of a node v at level i , v becomes the parent of u .

544 Assigning points to nodes requires careful attention. To guarantee that all clusters are disjoint as
545 specified by the condition (iii) of arranging nodes, we maintain a property that for each PIT, given any
546 two points $u, v \in Y_{i-1}$ which are in (the same or different) double-label nodes at level i , $d_X(u, v) > \frac{8\delta}{\varepsilon^{i-1}}$.
547 Here, we reuse the red-blue matching algorithm of Filtser and Le [15] to determine which pairwise
548 nodes could be placed in the same PIT while guaranteeing the three conditions above.

549 In the end, we obtain a collection of PITs \mathcal{T} , and the levels of PIT in \mathcal{T} are off from the levels of the
550 net tree by 1. At the leaf level, pairwise nodes are single-label, and these labels are exactly the points in
551 S . For a level $i > 0$, we construct pairwise nodes from the nodes of T at level $i-1$, and they could be
552 single-label or double-label depending on how we pair up points in Y_{i-1} . Some points in Y_{i-1} might not
553 appear in the labels of pairwise nodes at level i of a PIT. See Figure 3.

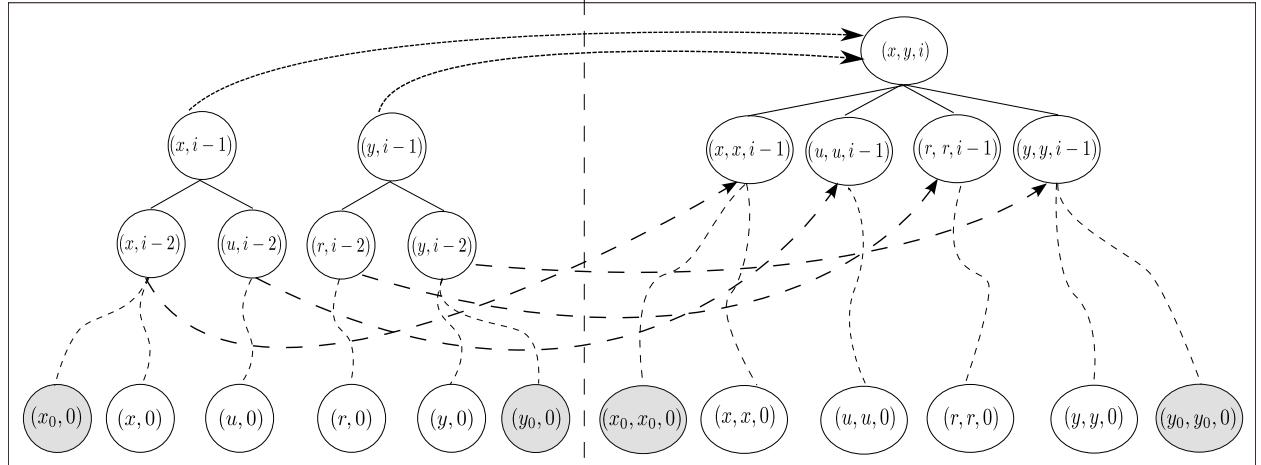


Figure 3: Illustrating a net tree T (left), a PIT T' derived from T (right). We create (x, y, i) by pairing up $(x, i-1)$ and $(y, i-1)$ in T , and this node is $O(\varepsilon)$ -close to the pair (x_0, y_0) . The dashed arrows show corresponding single-label pairwise nodes, and the dot arrows show corresponding double-label pairwise nodes.

554 Now, we show details of the static construction. We need the following lemma of Filtser and Le [16]:

555 **Lemma 1 ([16]).** Consider a graph $G = (V, E_b \cup E_r)$ that consists of disjoint edge sets called blue and
 556 red respectively. Let $\delta_r \geq 1$ ($\delta_b > 1$) be the maximal red (blue) degree. There exists a set \mathcal{M} of $O(\delta_b \delta_r)$
 557 maximal matchings such that: 1) their union covers all blue edges; 2) there is no red edges whose both
 558 endpoints are matched by any matching in \mathcal{M} .

559 The algorithm to construct \mathcal{M} of Filtser and Le [16] works roughly as follows. Let \mathcal{M} be the collection
 560 of maximal matchings, initially empty. Let B be the set of blue edges remaining uncovered in \mathcal{M} , initially
 561 $B = E_b$. We repeat the following process until B is empty: 1) greedily find M such that it is a maximal
 562 matching of B and there are no red edges whose endpoints are matched in M , 2) add M to \mathcal{M} , 3) remove
 563 edges in M out of B . We refer readers to the work of Filtser and Le [16] for the analysis of the properties
 564 \mathcal{M} . We call this algorithm **red-blue** matching.

565 3.1 The Static Construction

566 Given parameters $\delta > 0$, $\varepsilon < \frac{1}{16}$, a (δ, ε) -net tree T , our construction proceeds as follows. Initially, the
 567 collection has $\varepsilon^{-O(\lambda)}$ trees, each tree has a level 0 such that the set of leaf labels is a bijection with the
 568 point set S . We construct trees in T by visiting T in bottom-up order.

569 **[Step 1 - Create matchings]** To create pairwise nodes at level i for all trees in the collection, let
 570 Y_{i-1} be the set of all points at level $i-1$ of the net tree T . We define two important parameters: range
 571 $R_i = \left[(1-4\varepsilon) \frac{\delta}{\varepsilon^i}, (1+2\varepsilon) \frac{2\delta}{\varepsilon^i} \right]$, and threshold $s_i = \frac{10\delta}{\varepsilon^{i-1}}$. Let $G_i = (V_i, E_b \cup E_r)$ be the graph where the
 572 vertex set is $V_i = Y_{i-1}$, $E_b = \{(u, v) \in V_i \times V_i : d_X(u, v) \in R_i\}$, $E_r = \{(u, v) \in V_i \times V_i : d_X(u, v) < s_i\}$. This
 573 graph consists of blue edges (E_b) and red edges (E_r), where blue edges contain pairs of points in Y_{i-1}
 574 whose distances are in R_i , and red edges contain pairs whose distances are less than s_i . Since $\varepsilon < \frac{1}{16}$, we
 575 have $\frac{10\delta}{\varepsilon^{i-1}} < (1-4\varepsilon) \frac{\delta}{\varepsilon^i}$, which means an edge could be only red or blue. Applying the red-blue matching
 576 algorithm in Lemma 1, we obtain a set of matchings \mathcal{M}_i . See Figure 4.

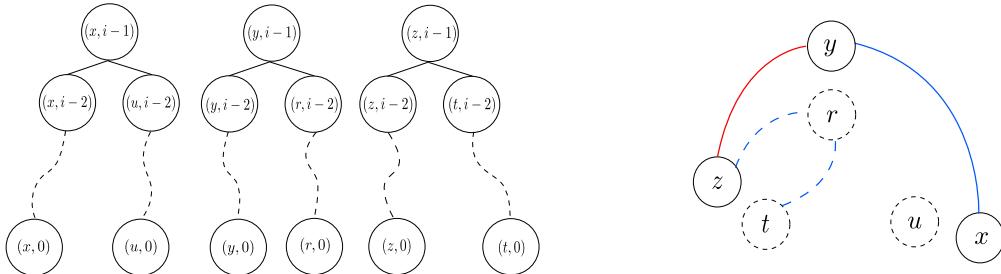


Figure 4: Examples of red-blue graphs created from Step 1. The left figure is the net tree T , and the right figure includes G_{i-1} and G_i . G_i consists of solid edges and nodes; G_{i-1} consists of dashed edges, and its vertex set includes dashed and solid nodes. The only blue edge in G_i is (x, y) , while G_{i-1} contains (r, z) and (r, t) as blue edges.

577 **[Step 2 - Create pairwise nodes at level i]** For each matching in \mathcal{M}_i , we will create pairwise nodes
 578 at level i for the corresponding PIT in T , then find children for these nodes as described below. Let M_j^i
 579 be the j^{th} matching of \mathcal{M}_i , and T'_j be the j^{th} PIT of T . Suppose by induction that we have already added
 580 pairwise nodes at level $i-1$ for all PITs in T from T nodes at level $i-1$. At this point, each PIT is a
 581 forest. We add pairwise nodes at level i to T'_j from M_j^i as follows:

582 • **[Step 2.1 - Matched nodes]** For each edge $(x, y) \in M_j^i$, we create a node (x, y, i) in T'_j , and
 583 then assign (x, y, i) as the **corresponding pairwise node** of $(x, i-1)$ and of $(y, i-1)$. If $i \geq 2$, for
 584 $u \in Y_{i-2}$ such that $d_X(u, x) \leq \frac{3\delta}{\varepsilon^{i-1}}$ or $d_X(u, y) \leq \frac{3\delta}{\varepsilon^{i-1}}$ (u might be x or y), let $(u_1, u_2, i-1)$ be the

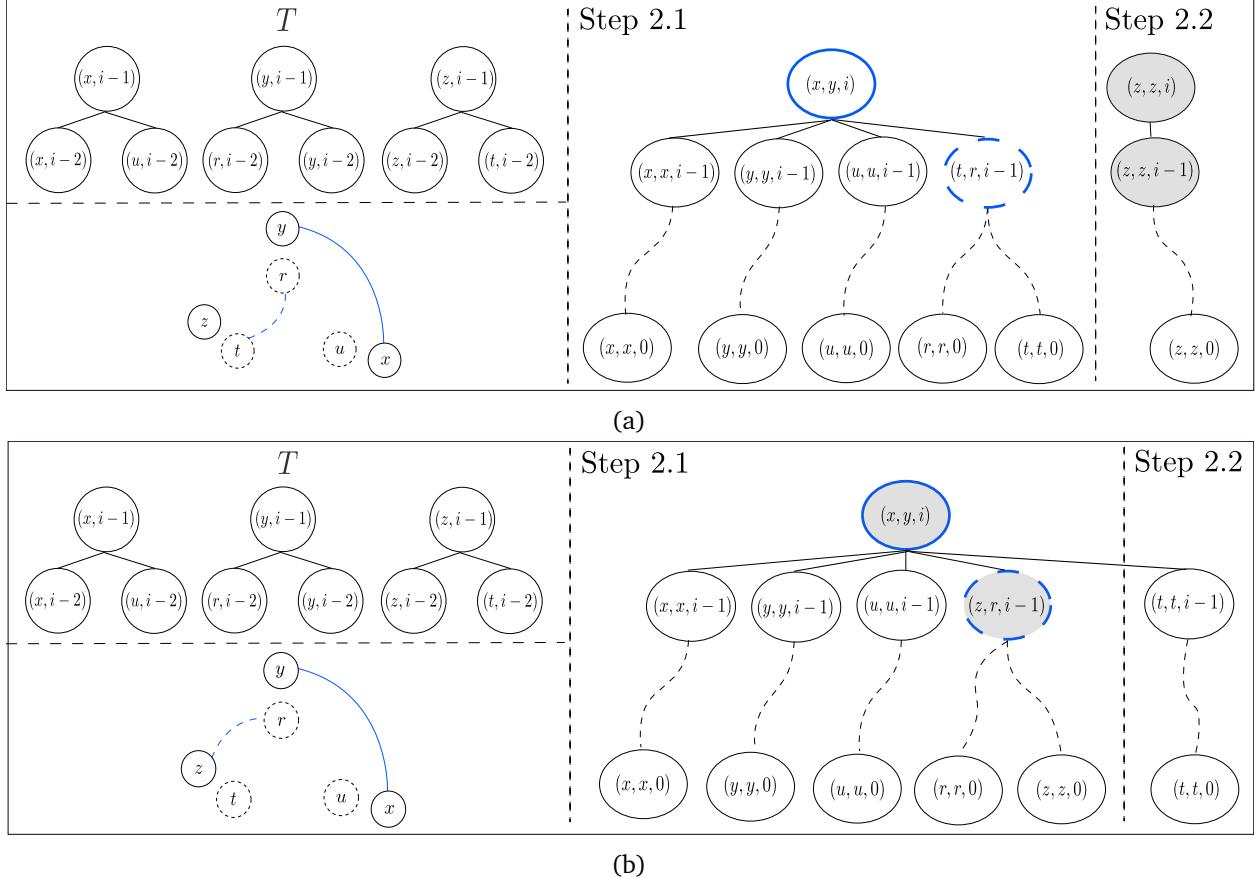


Figure 5: Illustration for step 2 - creating pairwise nodes. Depending on matched edges in G_i and pairwise nodes at level $i - 1$, we have different structures of PITs, as shown in figures (a) and (b). Blue lines are matched edges, and bold blue nodes are pairwise nodes created from these edges. The filled nodes are pairwise nodes of unmatched points, and specifically, (x, y, i) is the pairwise node of unmatched point $(z, i - 1)$.

585 corresponding pairwise node at level $i - 1$ of $(u, i - 2)$. We then set $(u_1, u_2, i - 1)$ to be a child
 586 of (x, y, i) . In Lemma 3 below, we show that $(u_1, u_2, i - 1)$ will not be set as a child of another
 587 pairwise node (x', y', i) created from another matched edge $(x', y') \in M_j^i$ using the fact that no
 588 red edge has both endpoints matched by M_j^i . See Figure 5 for an illustration.

589 Now, children of (x, y, i) include the corresponding pairwise nodes of $(x, i - 1)$'s children, $(y, i - 1)$'s
 590 children since if $(v, i - 2)$ is a child of $(x, i - 1)$ or $(y, i - 1)$ then $d_X(v, \{x, y\}) \leq \frac{\delta}{\varepsilon^{i-1}}$ by the covering
 591 property of T . Children of (x, y, i) might also contain corresponding pairwise nodes of children of
 592 some unmatched nodes.

593 • [Step 2.2 - Unmatched nodes] After going through all edges in M_j^i , we consider unmatched net
 594 point z in Y_{i-1} .

595 – [2.2.1] We create the corresponding pairwise node in T'_j for $(z, i - 1)$ by considering the
 596 corresponding pairwise node of $(z, i - 2)$:

597 (a) If $i = 1$, $(z, i - 2)$ does not exist, then we create (z, z, i) as the pairwise node of $(z, i - 1)$
 598 in T'_j .

599 (b) For $i > 1$, let $(z_1, z_2, i-1)$ be the corresponding pairwise node of $(z, i-2)$ in T'_j . If
600 $(z_1, z_2, i-1)$ does not have a parent, then we create (z, z, i) as the corresponding pairwise
601 node of $(z, i-1)$ in T'_j .
602 (c) If both (a) and (b) do not hold, meaning that $i \neq i$ and $(z_1, z_2, i-1)$ has a parent in T'_j ,
603 denoted by (z'_1, z'_2, i) , then we assign (z'_1, z'_2, i) as the corresponding pairwise node of
604 $(z, i-1)$ in T'_j .
605 - [2.2.2] For any child of $(t, i-2)$ of $(z, i-1)$ (this case only happens when $i > 1$), if the
606 corresponding pairwise node of $(t, i-2)$ in T'_j , denoted by $(t_1, t_2, i-1)$, has not been assigned
607 a parent, then we make $(t_1, t_2, i-1)$ a child of the corresponding pairwise node of $(z, i-1)$.
608 Otherwise, we leave $(t_1, t_2, i-1)$ as it is.

609 3.2 The Analysis

610 In this section, we will analyze the properties of the PITs constructed in the previous section, and
611 specifically, the packing and covering properties as defined in Definition 3 and the pairwise covering
612 property of the pairwise tree cover as defined in Definition 4. When $(p, i-1)$ is unmatched by a matching
613 and its level is clear from context, we will refer to p as an *unmatched point*. First, we observe that:

614 **Observation 1.** For any pairwise node (u_1, u_2, i) , $d_X(u_1, u_2) \leq (1 + 2\epsilon)^{\frac{2\delta}{\epsilon^i}}$.

615 **Proof:** If $u_1 = u_2$, then $d_X(u_1, u_2) = 0$. If $u_1 \neq u_2$, (u_1, u_2) must be a blue edge in G_i , thus $d_X(u_1, u_2) \leq$
616 $(1 + 2\epsilon)^{\frac{2\delta}{\epsilon^i}}$. \square

617 **Observation 2.** In step 2.2.1 case (c), $(z_1, z_2, i-1)$ already has a parent, which is found in step 2.1.

618 **Proof:** We consider all matched nodes first in step 2.1, then unmatched nodes later in step 2.2. In
619 each step, we find parents for some pairwise nodes at level $i-1$ in T'_j . In case (c) of step 2.2.1, the
620 corresponding pairwise node $(z_1, z_2, i-1)$ of $(z, i-2)$ already has a parent, which must be found by step
621 2.1. Thus the observation follows. \square

622 In the next two lemmas, we will show basic facts about the PITs.

623 **Lemma 2.** For any point $p \in Y_{i-1}$, let (u_1, u_2, i) be the corresponding pairwise node of p at level i in T'_j ,
624 then $d_X(p, \{u_1, u_2\}) = \min\{d_X(p, u_1), d_X(p, u_2)\} \leq \frac{6\delta}{\epsilon^{i-1}}$.

625 **Proof:** If the corresponding pairwise node at level i of $(p, i-1)$ is labeled by p , which means $p \in \{u_1, u_2\}$,
626 then $d_X(p, \{u_1, u_2\}) = 0$. We remain to consider the case $p \notin \{u_1, u_2\}$. This occurs in step 2.2.1 case (c),
627 when p is unmatched by M_j^i and the corresponding pairwise node of $(p, i-2)$, denoted by $(t_1, t_2, i-1)$,
628 has a node (u_1, u_2, i) as the parent for $i \geq 2$. Note that t_1 and t_2 could be the same or different points.
629 By Observation 2, $(t_1, t_2, i-1)$ become a child of (u_1, u_2, i) in step 2.1. Thus there exists $t \in Y_{i-2}$ such
630 that $d_X(t, \{u_1, u_2\}) \leq \frac{3\delta}{\epsilon^{i-1}}$ and $(t_1, t_2, i-1)$ is the corresponding pairwise node in T'_j of $(t, i-2)$. Now,
631 we prove that $d_X(p, \{u_1, u_2\}) \leq \frac{6\delta}{\epsilon^{i-1}}$. By induction, $d_X(p, \{t_1, t_2\})$ and $d_X(t, \{t_1, t_2\})$ are at most $\frac{6\delta}{\epsilon^{i-2}}$,
632 since $(t_1, t_2, i-1)$ is the corresponding pairwise node in T'_j of both $(p, i-2)$ and $(t, i-2)$. We have
633 $d_X(t_1, t_2) \leq (1 + 2\epsilon)^{\frac{2\delta}{\epsilon^{i-1}}}$ by Observation 1. Thus:

$$\begin{aligned}
d_X(p, t) &\leq d_X(p, \{t_1, t_2\}) + d_X(t_1, t_2) + d_X(\{t_1, t_2\}, t) \\
&\leq \frac{6\delta}{\epsilon^{i-2}} + (1 + 2\epsilon)^{\frac{2\delta}{\epsilon^{i-1}}} + \frac{6\delta}{\epsilon^{i-2}} \\
&= \frac{2\delta}{\epsilon^{i-1}} + \frac{16\delta}{\epsilon^{i-2}} < \frac{3\delta}{\epsilon^{i-1}} \quad (\text{since } \epsilon < \frac{1}{16})
\end{aligned}$$

635

Finally, by triangle inequality, we have:

636

$$\begin{aligned}
 d_X(p, \{u_1, u_2\}) &\leq d_X(p, t) + d_X(t, \{u_1, u_2\}) \\
 &\leq \frac{3\delta}{\varepsilon^{i-1}} + \frac{3\delta}{\varepsilon^{i-1}} \quad (d_X(t, \{u_1, u_2\}) \leq \frac{3\delta}{\varepsilon^{i-1}}) \\
 &= \frac{6\delta}{\varepsilon^{i-1}}
 \end{aligned}$$

637

This completes the proof. \square

638 **Lemma 3.** Consider step 2.1 of the construction, where we create pairwise nodes at level i for T'_j from
 639 edges matched in M_j^i . Let $(u_1, u_2, i-1)$ be a pairwise node at level $i-1$ of T'_j . There exists at most one
 640 matched blue edge $(x, y) \in M_j^i$ such that $(u_1, u_2, i-1)$ can be assigned as a child of the pairwise node
 641 (x, y, i) created from the blue edge (x, y) .

642 **Proof:** For contradiction, suppose there exists an edge $(x', y') \in M_j^i$ such that $(u_1, u_2, i-1)$ can be
 643 assigned as a child of both (x, y, i) and (x', y', i) . By step 2.1, $(u_1, u_2, i-1)$ can be a child of (x, y, i) if
 644 there exists $u \in Y_{i-2}$ such that $d_X(u, \{x, y\}) \leq \frac{3\delta}{\varepsilon^{i-1}}$, and $(u_1, u_2, i-1)$ is the corresponding pairwise node
 645 of $(u, i-2)$ in T'_j . Similarly, there exists $u' \in Y_{i-2}$ such that $d_X(u', \{x', y'\}) \leq \frac{3\delta}{\varepsilon^{i-1}}$, and $(u_1, u_2, i-1)$ is the
 646 corresponding pairwise node of $(u', i-2)$ in T'_j . First, we bound $d_X(u, u')$. By Lemma 2, $d_X(u, \{u_1, u_2\}) \leq$
 647 $\frac{6\delta}{\varepsilon^{i-2}}$ and $d_X(u', \{u_1, u_2\}) \leq \frac{6\delta}{\varepsilon^{i-2}}$. By Observation 1, $d_X(u_1, u_2) \leq (1 + 2\varepsilon) \frac{2\delta}{\varepsilon^{i-1}}$. We obtain:

$$\begin{aligned}
 d_X(u, u') &\leq d_X(u, \{u_1, u_2\}) + d_X(u_1, u_2) + d_X(\{u_1, u_2\}, u') \\
 &\leq \frac{6\delta}{\varepsilon^{i-2}} + (1 + 2\varepsilon) \frac{2\delta}{\varepsilon^{i-1}} + \frac{6\delta}{\varepsilon^{i-2}} \\
 &= \frac{2\delta}{\varepsilon^{i-1}} + \frac{16\delta}{\varepsilon^{i-2}}
 \end{aligned} \tag{1}$$

649 Since $d_X(u, \{x, y\}) \leq \frac{3\delta}{\varepsilon^{i-1}}$ and $d_X(u', \{x, y\}) \leq \frac{3\delta}{\varepsilon^{i-1}}$, and by triangle inequality, we have:

$$\begin{aligned}
 d_X(\{x, y\}, \{x', y'\}) &\leq d_X(\{x, y\}, u) + d_X(u, u') + d_X(u', \{x', y'\}) \\
 &\leq \frac{3\delta}{\varepsilon^{i-1}} + \frac{2\delta}{\varepsilon^{i-1}} + \frac{16\delta}{\varepsilon^{i-2}} + \frac{3\delta}{\varepsilon^{i-1}} \quad (\text{By Equation (1)}) \\
 &= \frac{8\delta}{\varepsilon^{i-1}} + \frac{16\delta}{\varepsilon^{i-2}} < \frac{10\delta}{\varepsilon^{i-1}} \quad (\text{since } \varepsilon < \frac{1}{16})
 \end{aligned}$$

651 It follows that there is a red edge connecting a point in $\{x, y\}$ and a point in $\{x', y'\}$. Since x and y are
 652 matched, (x', y') does not exist in M_j^i by the red-blue matching algorithm, contradicting the assumption
 653 that $(x', y') \in M_j^i$. \square

654 We are now ready to show the packing and covering of a PIT using Lemma 2 and Lemma 3 above.

655 **Lemma 4.** Each tree T'_j in the collection \mathcal{T}' satisfies packing and covering properties of PIT as defined
 656 in Definition 3: [packing] for any two nodes (x, y, i) and (u, v, i) , the distance between any pair of points
 657 in $\{x, y, u, v\}$ is $\Omega(\frac{\delta}{\varepsilon^{i-1}})$; [covering] (i) label points p in children of (x, y, i) has $d_X(p, \{x, y\}) \leq \frac{6\delta}{\varepsilon^{i-1}}$, and
 658 (ii) diameter of the cluster of a node at level i is bounded by $\frac{6\delta}{\varepsilon^{i-1}}$.

659 **Proof:** Since $x, y, u, v \in Y_{i-1}$, by the packing property of net tree T , the distance between any pair of
 660 points in $\{x, y, u, v\}$ is $\Omega(\frac{\delta}{\varepsilon^{i-1}})$. This implies that T'_j has the packing property of PITs.

661 To prove (i) of the covering, we consider a pairwise node $(u_1, u_2, i-1)$ of T'_j for $u_1, u_2 \in Y_{i-2}$ and its
 662 parent (x, y, i) for $x, y \in Y_{i-1}$.

663 If $(u_1, u_2, i-1)$ becomes a child of (x, y, i) in step 2.1, then $(x, y) \in M_j^i$ and there exists $u \in Y_{i-2}$
 664 such that $(u_1, u_2, i-1)$ is the pairwise node of $(u, i-2)$, and $d_X(u, \{x, y\}) \leq \frac{3\delta}{\varepsilon^{i-1}}$. Now we bound
 665 $d_X(u_1, \{x, y\})$ and $d_X(u_2, \{x, y\})$ by $d_X(u, \{u_1, u_2\})$. By Lemma 2, $d_X(u, \{u_1, u_2\}) \leq \frac{6\delta}{\varepsilon^{i-2}}$. By Observation 1,
 666 $d_X(u_1, u_2) \leq (1+2\varepsilon) \frac{2\delta}{\varepsilon^{i-1}}$. Thus:

$$\begin{aligned} d_X(u_1, \{x, y\}) &\leq d_X(u_1, u) + d_X(u, \{x, y\}) \\ &\leq d_X(u_1, u_2) + d_X(\{u_1, u_2\}, u) + d_X(u, \{x, y\}) \\ &\leq (1+2\varepsilon) \frac{2\delta}{\varepsilon^{i-1}} + \frac{6\delta}{\varepsilon^{i-2}} + \frac{3\delta}{\varepsilon^{i-1}} \\ &\leq \frac{6\delta}{\varepsilon^{i-1}} \quad (\text{since } \varepsilon < \frac{1}{16}) \end{aligned}$$

668 By the same argument, we get $d_X(u_2, \{x, y\}) \leq \frac{6\delta}{\varepsilon^{i-1}}$.

669 Now we consider when $(u_1, u_2, i-1)$ is assigned as a child of (x, y, i) in step 2.2. Observe that
 670 $(u_1, u_2, i-1)$ must be the corresponding pairwise node of $(t, i-2)$ where $(t, i-2)$ is a child of an
 671 unmatched node $(z, i-1)$. If $t = z$, we must create (z, z, i) as the corresponding pairwise node of $(z, i-1)$
 672 in step 2.2.1, thus $x = y = z$ and $d_X(t, \{x, y\}) = 0$. If $t \neq z$, the corresponding pairwise node of $(z, i-1)$
 673 has two cases:

- 674 • If $z = x = y$, which means we create (z, z, i) , then $d_X(t, \{x, y\}) = d_X(t, z) \leq \frac{\delta}{\varepsilon^{i-2}}$ by the covering
 675 property of T .
- 676 • If $z \neq x$ and $z \neq y$, then the corresponding pairwise node of $(z, i-2)$, say $(z_1, z_2, i-1)$, must be
 677 assigned as a child of (x, y, i) in step 2.1. This means $d_X(z, \{x, y\}) \leq \frac{3\delta}{\varepsilon^{i-1}}$. We have $d_X(t, z) \leq \frac{\delta}{\varepsilon^{i-2}}$
 678 by the covering property of T . Therefore, $d_X(t, \{x, y\}) \leq d_X(t, z) + d_X(z, \{x, y\}) \leq \frac{3\delta}{\varepsilon^{i-1}} + \frac{\delta}{\varepsilon^{i-2}}$.

679 In any case, we obtain $d_X(t, \{x, y\}) \leq \frac{3\delta}{\varepsilon^{i-1}} + \frac{\delta}{\varepsilon^{i-2}}$. By Lemma 2, $d_X(t, \{u_1, u_2\}) \leq \frac{6\delta}{\varepsilon^{i-2}}$. By Observation 1,
 680 $d_X(u_1, u_2) \leq (1+2\varepsilon) \frac{2\delta}{\varepsilon^{i-1}}$. Thus:

$$\begin{aligned} d_X(u_1, \{x, y\}) &\leq d_X(u_1, t) + d_X(t, \{x, y\}) \\ &\leq d_X(u_1, u_2) + d_X(t, \{u_1, u_2\}) + d_X(t, \{x, y\}) \\ &\leq (1+2\varepsilon) \frac{2\delta}{\varepsilon^{i-1}} + \frac{6\delta}{\varepsilon^{i-2}} + \frac{3\delta}{\varepsilon^{i-1}} + \frac{\delta}{\varepsilon^{i-2}} \\ &\leq \frac{6\delta}{\varepsilon^{i-1}} \quad (\text{since } \varepsilon < \frac{1}{16}) \end{aligned}$$

682 By the same argument, $d_X(u_2, \{x, y\}) \leq \frac{6\delta}{\varepsilon^{i-1}}$, giving item (i) of the covering property.

683 Finally, we bound the cluster-diameter. Recall that $C_i(x, y)$ is the set of leaves in the subtree rooted at
 684 (x, y, i) . We denote by $diam(C_i(x, y))$ the diameter of $C_i(x, y)$. To bound $diam(C_i(x, y))$, we consider
 685 $d_X(x, y)$, the distance from $\{x, y\}$ to labels of children of (x, y, i) , and the cluster-diameter of children
 686 nodes. Let $(u, v, i-1)$ be a child of (x, y, i) . We have $d_X(x, y) \leq (1+2\varepsilon) \frac{2\delta}{\varepsilon^i}$ by Observation 1, and
 687 $d_X(u, \{x, y\})$ and $d_X(v, \{x, y\})$ are at most $\frac{6\delta}{\varepsilon^{i-1}}$ by item (i) of the covering property of T'_j . By induction,
 688 suppose that $diam(C_{i-1}(u, v)) \leq \frac{6\delta}{\varepsilon^{i-1}}$. By triangle inequality, we obtain:

$$\begin{aligned} diam(C_i(x, y)) &\leq d_X(x, y) + 2 \cdot \max_{\text{a child } (u, v, i-1) \text{ of } (x, y, i)} \{d_X(\{u, v\}, \{x, y\}) + diam(C_{i-1}(u, v))\} \\ &\leq (1+2\varepsilon) \frac{2\delta}{\varepsilon^i} + 2 \left(\frac{6\delta}{\varepsilon^{i-1}} + \frac{6\delta}{\varepsilon^{i-1}} \right) \\ &= \frac{2\delta}{\varepsilon^i} + \frac{28\delta}{\varepsilon^{i-1}} < \frac{6\delta}{\varepsilon^i} \quad (\text{since } \varepsilon < \frac{1}{16}) \end{aligned}$$

690 This completes the proof. \square

691 Now we prove our main theorem of this section.

692 **Proof (Proof of Theorem 9):** Let \mathcal{T} be the collection of PITs obtained by running the static construction
693 in the previous algorithm to every (δ, ε) -net tree T with $\delta \in \{1, 2^1, 2^2, \dots, 2^{\lceil \lg(1/\varepsilon) \rceil}\}$. We have shown in
694 Lemma 4 that every PIT in \mathcal{T} satisfies the packing and covering property.

695 To bound the number of trees in \mathcal{T} , recall that for each level $i-1$ of the net tree T , we create a graph
696 G_i and run the matching algorithm. By Lemma 1, the algorithm returns \mathcal{M}_i with $|\mathcal{M}_i| = \varepsilon^{-O(\lambda)}$. We
697 create pairwise nodes of T'_j by the matching j^{th} of \mathcal{M}_i , thus \mathcal{T}' has $\varepsilon^{-O(\lambda)}$ PITs.

698 It remains to show the pairwise covering property of \mathcal{T} as defined in Definition 4. Consider two
699 points x_0, y_0 with $d_X(x_0, y_0) \in [\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i}]$ for an integer i . By the covering property of (δ, ε) -net tree, the
700 distance of a node at level $i-1$ to its children is at most $\frac{\delta}{\varepsilon^{i-1}}$. This implies the distance of a node at
701 level $i-1$ in T to its descendants is at most $\frac{2\delta}{\varepsilon^{i-1}}$. Therefore, there are two nodes $(x, i-1), (y, i-1)$
702 in T such that $d_X(x, x_0) \leq \frac{2\delta}{\varepsilon^{i-1}}$ and $d_X(y, y_0) \leq \frac{2\delta}{\varepsilon^{i-1}}$. By triangle inequality, $\|d_X(x, y) - d_X(x_0, y_0)\| \leq$
703 $d_X(x, x_0) + d_X(y, y_0)$. It follows that $d_X(x, y) \in [(1-4\varepsilon)\frac{\delta}{\varepsilon^i}, (1+2\varepsilon)\frac{2\delta}{\varepsilon^i}]$. By step 1 and step 2, there
704 must be a PIT $T' \in \mathcal{T}$ such that T' has a node (x, y, i) . Furthermore, $C_i(x, y)$ contains $B(x, \frac{2\delta}{\varepsilon^{i-1}})$ and
705 $B(y, \frac{2\delta}{\varepsilon^{i-1}})$. To see this, for any point t where $d_X(t, \{x, y\}) \leq \frac{2\delta}{\varepsilon^{i-1}}$, let $(u_1, u_2, i-1)$ be the pairwise node of
706 T' at level $i-1$ such that its cluster, $C_{i-1}(u_1, u_2)$, contains t . By Lemma 4, the diameter of $C_{i-1}(u_1, u_2)$ is
707 at most $\frac{6\delta}{\varepsilon^{i-2}}$, we have $d_X(t, u_1)$ and $d_X(t, u_2)$ are at most $\frac{6\delta}{\varepsilon^{i-2}}$. Therefore, $d_X(u_1, \{x, y\})$ and $d_X(u_2, \{x, y\})$
708 are at most $\frac{6\delta}{\varepsilon^{i-2}} + \frac{2\delta}{\varepsilon^{i-1}} < \frac{3\delta}{\varepsilon^{i-1}}$. By step 2.1, $(u_1, u_2, i-1)$ is a child of (x, y, i) , thus $C_i(x, y)$ contains
709 $C_{i-1}(u_1, u_2)$, which contains t whose $d_X(t, \{x, y\}) \leq \frac{2\delta}{\varepsilon^{i-1}}$.

710 We now prove that (x, y, i) is $O(\varepsilon)$ -close to the pair (x_0, y_0) . Consider a point $p \in C_i(x, y)$, we bound
711 $d_X(p, \{x_0, y_0\})$ by $d_X(p, \{x, y\})$ as follows. Observe that p is in a subtree rooted at a child $(u, v, i-1)$ of
712 (x, y, i) for some $u, v \in Y_{i-2}$. By Lemma 4, the diameter $C_{i-1}(u, v)$ is bounded by $\frac{6\delta}{\varepsilon^{i-1}}$, thus $d_X(p, u) \leq \frac{6\delta}{\varepsilon^{i-1}}$
713 and $d_X(p, v) \leq \frac{6\delta}{\varepsilon^{i-1}}$. By item (i) in the covering property of T' (Lemma 4), we have $d_X(u, \{x, y\})$ and
714 $d_X(v, \{x, y\})$ are at most $\frac{6\delta}{\varepsilon^{i-1}}$. Therefore:

$$\begin{aligned} d_X(p, \{x, y\}) &\leq \min\{d_X(p, u) + d_X(u, \{x, y\}), d_X(p, v) + d_X(v, \{x, y\})\} \\ &\leq \frac{6\delta}{\varepsilon^{i-1}} + \frac{6\delta}{\varepsilon^{i-1}} = \frac{12\delta}{\varepsilon^{i-1}} \end{aligned} \tag{2}$$

716 It follows that:

$$\begin{aligned} d_X(p, \{x_0, y_0\}) &\leq \min\{d_X(p, x) + d_X(x, x_0), d_X(p, y) + d_X(y, y_0)\} \\ &\leq d_X(p, \{x, y\}) + \frac{2\delta}{\varepsilon^{i-1}} \quad (\text{since } d_X(x, x_0) \leq \frac{2\delta}{\varepsilon^{i-1}} \text{ and } d_X(y, y_0) \leq \frac{2\delta}{\varepsilon^{i-1}}) \\ &\leq \frac{14\delta}{\varepsilon^{i-1}} \end{aligned} \tag{3}$$

718 Since $d_X(x_0, y_0) \geq \frac{\delta}{\varepsilon^i}$, we obtain $d_X(p, \{x_0, y_0\}) \leq 14\varepsilon d_X(x_0, y_0)$ as claimed. \square

719 **Remark 1.** The PITs constructed in Section 3.1 may not have the hierarchical property in the sense that
720 the points labeling a node u at level i may not be a subset of points labeling the children of u . We can
721 enforce this hierarchical property by renaming the labels as follows. First, we claim that given a pairwise
722 node (x, y, i) of a PIT T' , children labels of (x, y, i) can be partitioned into two disjoint sets S_1 and S_2 of
723 diameter $\Theta(\frac{\delta}{\varepsilon^{i-1}})$, S_1 is close to x and S_2 is close to y (if $x = y$ then S_2 is empty). Observe that this claim
724 follows by two items of the covering property of T' : for any child $(u_1, u_2, i-1)$ of (x, y, i) , $d_X(u_1, \{x, y\})$
725 and $d_X(u_2, \{x, y\})$ are at most $\frac{6\delta}{\varepsilon^{i-1}}$, and the diameter of $C_i(x, y)$ is at most $\frac{6\delta}{\varepsilon^i}$. By the packing property

726 of T , any $u, v \in Y_i$ has $d_X(u, v) > \frac{\delta}{\epsilon^i}$, thus $S_1 \cap Y_i$ (and in $S_2 \cap Y_i$) has at most one point. Therefore,
 727 there are at most two points in Y_i that are also in children labels of (x, y, i) . Now, whenever we create
 728 a pairwise node (x, y, i) , we find the corresponding pairwise nodes $(x_1, x_2, i-1)$ and $(y_1, y_2, i-1)$ in
 729 T' of $(x, i-2)$ and $(y, i-2)$. If $x_1 = x_2$ and $x \neq x_1$, we rename $(x_1, x_2, i-1)$ to $(x, x, i-1)$. If $x \neq x_1$
 730 and $x \neq x_2$, we rename $(x_1, x_2, i-1)$ to $(x, x_2, i-1)$ if $d_X(x, x_1) \leq d_X(x, x_2)$, to $(x_1, x, i-1)$ otherwise.
 731 Similarly, if $y_1 = y_2$ and $y \neq y_1$, we rename $(y_1, y_2, i-1)$ to $(y, y, i-1)$. If $y \neq y_1$ and $y \neq y_2$, we
 732 rename $(y_1, y_2, i-1)$ to $(y, y_2, i-1)$ if $d_X(y, y_1) \leq d_X(y, y_2)$, to $(y_1, y, i-1)$ otherwise.

733 4 Pairwise Tree Cover: Dynamic Construction

734 In this section, we construct a data structure for maintaining a dynamic pairwise tree cover for a point
 735 set under updates as claimed in Theorem 2, which we restate below.

736 **Theorem 2.** *Given $\epsilon > 0$, there is a data structure maintaining (τ, ϵ) -pairwise tree cover with $\tau = \epsilon^{-O(\lambda)}$
 737 supporting INSERT/DELETE in $O(\epsilon^{-O(\lambda)} \log(n))$ time per operation, and GETPREDECESSOR/GETSUCCESSOR
 738 in $O(1)$ time per operation.*

739 In Section 3, we outlined how a collection of PIT can be statically derived from a net tree. The static
 740 construction assumes the full net tree T where the net points at every level are given explicitly. However,
 741 such a full net tree would have size $\Omega(n \log \Delta)$. In dynamic construction, we cannot afford to maintain
 742 every level of T explicitly. Instead, we need to maintain a *compressed net tree*, for every level i , some
 743 nodes will be hidden (and hence can only be accessed indirectly) to guarantee that the total size is $O(n)$.

744 **Dynamic compressed net tree.** Nodes at some level i of the (uncompressed) net tree will be hidden
 745 via jumps: A *jump* is an edge in T connecting a node (x, h) at level h and a node (x, l) at a lower level l
 746 where $l < h - 1$. The jump from (x, h) down to (x, l) effectively hides all level- i nodes (x, i) for every
 747 $l < i < h - 1$; we call such a node (x, i) a *hidden node*. We call (x, h) the *top* of the jump and (x, l) the
 748 *bottom* of the jump. For a technical reason, we will maintain that every jump in T starting from a node
 749 (x, h) down to (x, l) will be *b-isolated*: given a jump, for any node (y, k) who is not a descendant of
 750 (x, h) for $k < h$, $d_X(x, y) > b \frac{\delta}{\epsilon^k}$. Furthermore, in a dynamic net tree, nodes are marked deleted rather
 751 than explicitly deleted; we will elaborate more details by the end of this section. (Herein, we use the
 752 term dynamic net tree to refer to the dynamic compressed net tree.) Note that the compressed net tree
 753 still has a degree bounded by $\epsilon^{-O(\lambda)}$, since the packing and covering properties still hold.

754 **Definition 7 ((δ, ϵ)-dynamic net tree).** *(δ, ϵ) -dynamic net tree is a data structure maintaining a (δ, ϵ) -
 755 net tree T under insertions and deletions. The data structure supports the following operations:*

- 756 • **INSERT(p, T):** *Insert (possibly more than one) nodes at different levels associated with a new point
 757 p to T , and return a list of $O_\lambda(1)$ new nodes or nodes whose parents in T are updated due to
 758 inserting p . There are three types of nodes in the list:*
 - 759 1. **new-point node:** *when a new point p is added to S , up to three new-point nodes associated
 760 with p might be created: $(p, 0)$, (p, i) , and $(p, i-1)$ for some level $i > 0$. Node (p, i) is created
 761 as a new child of some node $(u, i+1)$ in T , and furthermore, it will be the top of the jump
 762 down to $(p, 0)$, making $(p, 0)$ the only child of (p, i) . Once $(p, 0)$ and (p, i) are created, the
 763 algorithm might additionally create $(p, i-1)$ as a node between $(p, 0)$ and (p, i) (to split the
 764 jump from (p, i) down to $(p, 0)$) for maintaining the jump isolation property.*
 - 765 2. **splitting-jump node:** *which is new node (q, i) added at the middle of the jump from (q, l)
 766 down to (q, h) for $l < i < h$.*

767 3. *promoting node*: which is a new node (q, i) in T created by applying an operation called
 768 PROMOTE($q, i - 1, T$) to the node $(q, i - 1)$ at level $i - 1$. As a result of this operation, the
 769 parent in T of $(q, i - 1)$ was changed from some node (u, i) , with $u \neq q$, at level i to (q, i) .
 770 Furthermore, another node at level $i + 1$ will be designated as the parent of (q, i) . We call
 771 $(q, i - 1)$ a *promoted* node, as the point q was “promoted” to level i from level $i - 1$.

772 • DELETE(p, T): mark the leaf of p as deleted and return the pointer to the leaf.

773 In Section 6, we review and slightly simplify the dynamic net tree construction of Cole and Gotlieb [12].
 774 Readers who are not familiar with the work of Cole and Gotlieb [12] are strongly encouraged to read
 775 Section 6 to have a complete understanding of how a dynamic net tree changes under updates. Our
 776 *intuition* for constructing other data structures will be built on top of the dynamic net tree. However,
 777 our technical proofs presented here will only rely on the facts stated in the following theorem, whose
 778 proof will be given in Section 6. The key properties are packing and covering; the jump isolation and
 779 close-containment properties are needed for technical purposes only.

780 **Theorem 10.** *Given $b \geq 2$ a parameter of the jump isolation, $\varepsilon \leq \frac{1}{4b}$, there is a data structure maintaining
 781 a (δ, ε) -net tree T such that T has the following properties:*

782 • [Packing.] Two nodes at the same level (x, i) and (y, i) have $d_X(x, y) > \frac{1}{4} \frac{\delta}{\varepsilon^i}$.
 783 • [Covering.] If (x, i) is the parent of (y, i') where $i' < i$, then $d_X(x, y) \leq \phi \frac{\delta}{\varepsilon^i}$, where $\phi = \frac{3}{4}$.
 784 • [b-Jump isolation.] Any jump is *b-isolated*: given a jump starting from a node (x, i) , for any node
 785 (y, k) who is not a descendant of (x, i) for $k < i$, $d_X(x, y) > b \frac{\delta}{\varepsilon^k}$.
 786 • [Close-containment.] For any (y, k) and any ancestor (z, i) of (y, k) , $d_X(y, z) \leq \frac{\delta}{\varepsilon^i} - \frac{\delta}{\varepsilon^k}$. This
 787 implies that every point p in the subtree rooted at (z, i) is contained in $B(z, \frac{\delta}{\varepsilon^i})$, i.e., $d_X(p, z) \leq \frac{\delta}{\varepsilon^i}$.

788 Furthermore, given access to a node (x, i) in T at level i , if (x, i) is not the bottom node or a hidden node
 789 in a jump, then we can find all the nodes (y, i) at level i such that $d_X(x, y) = g \cdot \frac{\delta}{\varepsilon^i}$ for any parameter
 790 $g \geq 1$ in $O(g)^\lambda$ time. The data structure has space $O(n)$ and runs in $O_\lambda(\log n)$ time per update.

791 Since not all the nodes are explicitly accessible in a compressed net tree, the construction of a PIT
 792 from a compressed net tree is somewhat cumbersome. The key observation to keep in mind is the locality
 793 of our static construction in Section 3; specifically, the neighborhood of a node in a PIT is a subset of the
 794 neighborhood of the corresponding node in a net tree T . This locality alone allows one to maintain a
 795 dynamic PIT from a dynamic net tree T .

796 The much more difficult task is to maintain a dynamic leaf ordering of PIT due to two key challenges
 797 (C1) and (C2) outline in Section 1.1. We accomplish this task in several steps; see the block diagram
 798 in Figure 1 for an overview. The first step we take is to stabilize the net tree using a net tree cover as
 799 defined in Definition 8 below. (The formal definition is somewhat involved; we will briefly describe the
 800 idea afterward.) In this definition, to distinguish nodes between different trees, we denote a *node* (u, i)
 801 *of T by (u, i, T)* , and a node (u, i) of J by (u, i, J) . When the context is clear about which tree is used, we
 802 simply denote a node by (u, i) .

803 **Definition 8 (Net tree cover).** *Let T be a (δ, ε) -net tree of a point set S in a doubling metric with
 804 dimension λ , and $c \geq 4$ be a constant parameter. Given $\varepsilon \leq \frac{1}{20}$, a (δ, ε) -net tree cover of T is a collection
 805 of trees $\mathcal{J} = \{J_1, J_2, \dots\}$ such that:*

806 • [Size.] $|\mathcal{J}| = O_\lambda(1)$ trees.

- [Net.] For each tree $J \in \mathcal{J}$, points at level $i + 1$ of J is an $O(\frac{\delta}{\epsilon^i})$ -net of S , where the set of nodes at level $i + 1$ of J is a subset of nodes at level i of T . Specifically, there exists a **surjective map** ψ_J that maps a node of T to a node of J , $\psi_J(x, i) = (w, i + 1, J)$, where w can be x or a different point, and for any node $(w, i + 1, J)$ in J where $i \geq 0$, there exists (w, i, T) in T such that $\psi_J(w, i) = (w, i + 1, J)$.
- [Partial isomorphism.] In every $J \in \mathcal{J}$, consider a node $(x, i + 1, J)$ where $i \geq 0$. If (x, i, T) does not have a parent update except by splitting a jump, then $(x, i + 1, J)$ is a child of $\psi_J(u, i')$ where (u, i', T) is the parent of (x, i, T) . We say that J is **partially isomorphic** to T .
- [Shifting.] For every pair of node (x, i) and (y, i) in T with $d_X(x, y) < \frac{c \cdot \delta}{\epsilon^{i+1}}$, there exists a tree $J \in \mathcal{J}$ such that $\psi_J(x, i) = (x, i + 1, J)$, $\psi_J(y, i) = (y, i + 1, J)$ and they have the same parent.
- [Pairwise covering.] For every pair of points $x_0, y_0 \in S$ such that $d_X(x_0, y_0) \in [\frac{\delta}{\epsilon^i}, \frac{2\delta}{\epsilon^i}]$, there exists a tree J such that $(x_0, 0, J)$ and $(y_0, 0, J)$ have the same ancestor at level $i + 1$.

The basic idea of net tree cover is to start from a net tree T , construct a constant number of trees in a set \mathcal{J} where the net points in level in each tree $J \in \mathcal{J}$ is a subset of net points in the corresponding level of T . Therefore, each tree in \mathcal{J} in some sense resembles T ; this is formalized in the net and partial isomorphism properties in Definition 8. As a (δ, ϵ) -net tree “takes care” of distances in the ranges $[\frac{\delta}{\epsilon^i}, \frac{2\delta}{\epsilon^i}]$, the cover \mathcal{J} also has to take care of these distances; this explains the covering property. The shifting property, on the other hand, captures the intuition that the cover \mathcal{J} was constructed by the shifting technique similar to grid shifting in Euclidean spaces [22, 10]. (We can fix the constant c in the shifting property to be 4, but this leads to a somewhat artificial-looking bound.) While the definition of tree cover is more complicated and somewhat unnatural, we are able to show that updates to T due to an insertion of a point to S can be decomposed into two types of very simple updates to a tree in \mathcal{J} : leaf insertions or edge subdivisions. We say that these updates are **stable**. For a technical reason, we need the dynamic net tree T to have the $3c$ -jump isolation property by simply setting $b = 3c$ in Theorem 10.

Theorem 11 (Dynamic Net Tree Cover). *Let T be a dynamic (δ, ϵ) -net tree for a dynamic point set S such that every jump in T is $3c$ -isolated where c is the constant in Definition 8. Then we can construct a dynamic net tree cover \mathcal{J} from T such that the updates T due to the insertion of a point to S induce $O_\lambda(1)$ updates to every tree $J \in \mathcal{J}$ that are stable: they contain $O(1)$ leaf insertions and $O(1)$ edge subdivisions. Furthermore, the updates to \mathcal{J} can be identified in $O_\lambda(1)$ time.*

Next, we will construct a dynamic PIT from a net tree cover \mathcal{J} . We call every tree $J \in \mathcal{J}$ a **stable (δ, ϵ) -net tree**. We basically follow the static construction in Section 3 to construct a collection of PITs from each stable tree J . As we noted earlier, the construction is local, and hence, whenever a new node (u, i) is inserted into J , we will develop a **dynamic pairing algorithm** to examine the local neighborhood of (u, i) to find nodes that can be paired up with (u, i) , and then update the corresponding PIT. As the net tree J is stable, we could guarantee that the dynamic PITs constructed from J by our dynamic pairing algorithm are also stable.

Definition 9 (Stable Dynamic PIT). *A **stable dynamic** PIT is a PIT that is under three types of updates: adding (a null or non-null) leaf, subdividing an edge, and marking a leaf as deleted.*

In the theorem below, we summarize the guarantees by our dynamic pairing algorithm. The proof will be given in Section 4.2.

Theorem 12 (Dynamic Pairing). *Let \mathcal{J} be a dynamic stable (δ, ϵ) -net tree cover constructed from a (δ, ϵ) -net tree in Theorem 11. Then we can construct from \mathcal{J} a collection of **stable** dynamic PITs \mathcal{T} such that (i) $|\mathcal{T}| = \epsilon^{-O(\lambda)}$ and (ii) for every points $x, y \in S$ where $d_X(x, y) \in [\frac{\delta}{\epsilon^i}, \frac{2\delta}{\epsilon^i}]$, there exists a PIT $T' \in \mathcal{T}$ such that a node at level i of T' is $O(\epsilon)$ -close to (x, y) . Furthermore, every update to a tree in \mathcal{J} can be translated into $\epsilon^{-O(\lambda)}$ updates to \mathcal{T} that can be identified in $\epsilon^{-O(\lambda)}$ time.*

Once we have a stable dynamic PIT, we could develop a data structure to keep track of the DFS ordering of its leaves. Note that the DFS visits nodes in the PIT by order of insertion time: to break ties between children of a node, the DFS will first visit those that were inserted earlier. Recall that the major issue in maintaining DFS leaf ordering of an unstable PIT is that when a node u changes its parent to a new node, the DFS ordering, as well as the subtrees, of all ancestors of u change significantly, and there could be up to $\Omega(n)$ such ancestors.

In a stable PIT, the parent of a node could also change but only in edge subdivision. Specifically, a node z could be inserted between a node u and its parent v , so that the parent of u now changes from v to z . However, the DFS leaf ordering of ancestors of u (except z) *does not change* due to the subdivision by z . (For z , the DFS leaf ordering in its subtree is exactly that of u since u is its only child.)

On the other hand, inserting a new leaf x to a PIT could still induce changes in the DFS leaf orderings of all (and up to $\Omega(n)$) ancestors of x . The key difference to the case of unstable PIT is that only a *single node* is inserted, and hence, the DFS leaf orderings and the subtrees of these ancestors only change by one, making it possible to use some lazy data structures. Here, we use a data structure by Kopelowitz and Lewenstein [25] to maintain a (dynamic) centroid decomposition on top of our stable PIT. The basic idea is that in a centroid decomposition, we could identify $O(\log n)$ important ancestors of each node (which are the top endpoint of centroid paths) such that it suffices to update these ancestors only.

A technical difficulty is that a stable PIT could have null leaves or leaves that are marked deleted. We say that a leaf is *active* if it is non-null and not marked deleted; otherwise, the leaf is *inactive*. We say that a node in a PIT is *active* if it has at least one active leaf in its subtree; otherwise, the node is *inactive*. In a DFS leaf ordering, we only keep track of active leaves. Here, stability also helps us in the following way: imagine that we iteratively contract an inactive node to its parent to obtain a tree of active nodes and leaves only. The edge subdivision does not really change the contracted tree by much, and hence, we could keep track of the DFS leaf ordering of the contracted tree. While the idea is rather simple, explicitly contracting nodes is expensive since inserting a new (active) leaf could turn a long chain of inactive ancestors to become active. Indeed, we only use contraction as a metaphor to develop our data structure; we do not really contract inactive nodes. All of these ideas lead to a leaf tracker data structure as defined formally below.

Definition 10. *Leaf tracker is a data structure that maintains a stable dynamic PIT T and a DFS-leaf ordering σ of active leaves of T and supports the following operations:*

- $\text{INSERTLEAF}(u, v, T)$: *insert the node v as a leaf under node u .*
- $\text{SUBDIVIDEEDGE}(v, e, T)$: *insert the node v such that v breaks an existing edge $e = (x, y)$ into two new edges (x, v) and (v, y) .*
- $\text{DELETELEAF}(u, T)$: *mark the leaf u as deleted.*
- $\text{TRACKLEFTMOSTLEAF}(u, T)$: *return the left-most leaf of a node u .*
- $\text{TRACKRIGHTMOSTLEAF}(u, T)$: *return the right-most leaf of a node u .*
- $\text{GETPREDECESSOR}(p, T)$: *return the predecessor of an active leaf p in σ .*
- $\text{GETSUCCESSOR}(p, T)$: *return the successor of an active leaf q in σ .*

The next theorem shows how to maintain a leaf tracker data structure efficiently; the proof will be given in Section 5.

Theorem 13. *We can construct a leaf tracker data structure for maintaining a stable dynamic PIT T with $O(\varepsilon^{-O(\lambda)} + \log(n))$ time per updating and tracking operation (including INSERTLEAF , SUBDIVIDEEDGE , DELETELEAF , TRACKLEFTMOSTLEAF , $\text{TRACKRIGHTMOSTLEAF}$). Furthermore, the DFS leaf ordering of T*

894 will be maintained in a doubly linked list, and hence the data structure could support $O(1)$ time per
 895 query (including GETPREDECESSOR, GETSUCCESSOR).

896 **About deletion.** As we mentioned at the beginning of this section, deletions are only marked: whenever
 897 a point p is deleted from S , we mark the leave nodes corresponding to p (in dynamic net tree, net tree
 898 cover, and PITs) to be deleted. We do not explicitly delete these nodes. The standard idea to handle these
 899 is that when the number of deletions is above a certain threshold, we will rebuild the data structure. The
 900 rebuilding leads to a data structure with amortized running time, and one can de-amortize by rebuilding
 901 in the background. All of these ideas were used by Cole and Gottlieb [12] to handle deletions in their
 902 dynamic net tree, and we follow exactly the same strategy to handle deletions.

903 Given all data structures, including net tree cover, dynamic pairing, and leaf tracker, we are now
 904 ready to prove Theorem 2.

905 **Proof (Proof of Theorem 2):** First, we apply Theorem 10 to maintain $O(\log \frac{1}{\varepsilon})$ dynamic (δ, ε) -net trees
 906 T for every $\delta \in \{1, 2, \dots, 2^{\lceil \lg 1/\varepsilon \rceil}\}$. Second, we maintain a dynamic net tree cover \mathcal{J} for each (δ, ε) -net
 907 tree T by applying Theorem 11; we can choose $c = 4$. Third, we apply the dynamic pairing algorithm in
 908 Theorem 12 to maintain a collection of PITs \mathcal{T} from \mathcal{J} . Observe that the total number of PITs constructed
 909 in this process for $O(\log \frac{1}{\varepsilon})$ different net trees is $\varepsilon^{-O(\lambda)}$. Finally, we maintain a leaf tracker data structure
 910 for each tree in \mathcal{T} to keep track of the DFS leaf ordering and querying the predecessor/successor of every
 911 active leaf.

912 Now, we analyze the update time and query time. Observe that the query time is $O(1)$ by Theorem 13.
 913 For the update, we note that the total update time to a net tree is $\varepsilon^{-O(\lambda)} \log n$ by Theorem 10. This also
 914 translates to $\varepsilon^{-O(\lambda)} \log n$ total update time to the net tree cover by Theorem 11; the same update time
 915 holds for each PIT by Theorem 12. Since the total number of PITs is $O_{\lambda, \varepsilon}(1)$, the total update times to all
 916 PITs is $\varepsilon^{-O(\lambda)} \log n$. By Theorem 13, the update time to the leaf tracker is $O(\varepsilon^{-O(\lambda)} + \log n)$. Thus, the
 917 final update time remains $\varepsilon^{-O(\lambda)} \log n$. \square

918 In Section 4.1 we construct a dynamic net tree cover to stabilize the dynamic net tree. In Section 4.2
 919 we give the details of the dynamic pairing algorithm. The leaf tracker data structure is rather complicated
 920 and will be given in Section 5.

921 4.1 Dynamic Net Tree Cover

922 In this section, we describe the ideas for stabilizing a dynamic net tree using a dynamic net tree cover as
 923 claimed in Theorem 11, which we restate here for convenience.

924 **Theorem 11 (Dynamic Net Tree Cover).** Let T be a dynamic (δ, ε) -net tree for a dynamic point set S
 925 such that every jump in T is $3c$ -isolated where c is the constant in Definition 8. Then we can construct a
 926 dynamic net tree cover \mathcal{J} from T such that the updates T due to the insertion of a point to S induce $O_{\lambda}(1)$
 927 updates to every tree $J \in \mathcal{J}$ that are stable: they contain $O(1)$ leaf insertions and $O(1)$ edge subdivisions.
 928 Furthermore, the updates to \mathcal{J} can be identified in $O_{\lambda}(1)$ time.

929 Recall that the only non-trivial parent update to a node (p, i) in a tree J in the cover \mathcal{J} is the edge
 930 subdivision: inserting a new node between (p, i) and its parent in J . Indeed, this subdivision is jump
 931 splitting defined in Definition 6; that is, the parent of (p, i) is (p, j) associated with the same point p at
 932 some level $j > i$. On the other hand, the only non-trivial parent update in a dynamic net tree T is due to
 933 promotion operation. Specifically, $\text{PROMOTE}(p, i-1, T)$ changes the parent of $(p, i-1)$ from some node
 934 (u, i) , with $u \neq p$, at level i to a newly created node (p, i) associated with p .

935 To stabilize the parent change due to $\text{PROMOTE}(p, i-1, T)$, the key observation is that $d_X(p, u)$ is
 936 small; precisely, $d_X(p, u) = O(\frac{\delta}{\varepsilon^{i-1}})$. As a thought experiment, suppose that we have a version of T ,
 937 denoted by J , where we only keep a subset of Y_i , say a $10\frac{\delta}{\varepsilon^i}$ -net⁴ of Y_i , at level i of J , and all other
 938 nodes in Y_i are discarded. (If a node (x, i) is discarded if it is too close to a node (y, i) that is kept in
 939 J , then children of (x, i) in T will become children of (y, i) in J .) Back to $\text{PROMOTE}(p, i-1, T)$, in an
 940 *ideal situation*, if some node (t, i) is kept in a version J of T where t is close enough to both u and p ,
 941 then: (1) node (u, i) will be discarded—we say that (u, i) is *merged by distance* to (t, i) to emphasize
 942 that (u, i) is discarded due to (t, i) —(2) there is no need to make a parent (p, i) of $(p, i-1)$ since (p, i)
 943 will also be discarded, and (3) node $(p, i-1)$ is already a child of (t, i) before the promotion due to
 944 the merge of (u, i) to (t, i) , and hence no parent update is needed. Of course, the difficulty here is that
 945 there is no good way to choose a $10\frac{\delta}{\varepsilon^i}$ -net of Y_i so that the ideal situation always happens. Nevertheless,
 946 this thought experiment leads us to the idea of using more than one tree: we simply partition Y_i into (a
 947 small) number of $10\frac{\delta}{\varepsilon^i}$ -nets, and for each net, construct a version J of T such that the ideal situation will
 948 happen in at least one of the tree. This is exactly the shifting in Definition 8. However, shifting alone is
 949 not enough to achieve stability: as points arrive dynamically, some existing nodes could be merged to a
 950 newly inserted node, leading to parent updates of the children of the existing nodes. We then introduce
 951 the idea of *merging through time* to handle this case. To expand on all ideas in this paragraph, we briefly
 952 describe how to achieve shifting and then stability.

953 **Shifting.** Observe that if two nodes $(x, i-1)$ and $(y, i-1)$ are relatively close, $d_X(x, y) = O(\frac{\delta}{\varepsilon^i})$, then
 954 their parents (u, i) and (v, i) are also close: $d_X(u, v) = O(\frac{\delta}{\varepsilon^{i+1}})$. As we mentioned above, the basic idea is
 955 to partition Y_i into Δ_i -net where $\Delta_i = 6c\frac{\delta}{\varepsilon^i}$ using a standard coloring trick, see, e.g., [3, 24]. We greedily
 956 color points in Y_i : when considering a new point y , we color y by the smallest available color that is
 957 different from the colors of the nodes within distance at most Δ_i from y . (Our actual coloring procedure
 958 is slightly different for a technical reason described below, but the idea is largely the same.) By packing
 959 bound, the number of colors is $O_{\lambda, \varepsilon}(1)$, and each color class will induce a Δ_i -net of Y_i .

960 Points in each color class, called *centers*, will be at level $i+1$ of a tree J in \mathcal{J} . The difference in one
 961 level between T and J is because the level 0 of J must form a bijection into S , but from level 0 of T
 962 we only have a subset of points. For non-center point (v, i) , if the distance from a center (u, i) is small
 963 enough (at most $\Delta_i/2$), then it might be *merged* to (u, i) in J in the sense that children of (v, i) in T will
 964 become children of (u, i) in J . More formally, if (v, i) is merged to (u, i) in J , then for any child $(t, i-1)$
 965 of (v, i) , $\psi_J(t, i-1)$ will be a child of $\psi_J(u, i)$. Not every non-center node will be merged to some center
 966 node; if they are far from any of the centers, then they will be left unmerged and will also appear at
 967 level $j+1$ in J .

968 A very subtle technical problem is that when we merge a non-center node (v, i) into a center node
 969 (u, i) , we would want any child $(w, i-1)$ of (v, i) to become a child of (u, i) . However, it is conceivable
 970 that $(w, i-1)$ is a non-center node at level $i-1$ and hence was merged to some other center node
 971 $(t, i-1)$, which is not a child of either (u, i) or (v, i) , and hence $(w, i-1)$ does not become a child of
 972 (u, i) . We fix this problem by using two *disjoint* sets of colors for any two consecutive levels of T .

973 The key guarantee we obtained from shifting is that for any nearby pair of net points, there exists a
 974 tree $J \in \mathcal{J}$ where the corresponding nodes in J of the two net points have the same parent.

975 **Achieving stability.** Observe that updates to T could induce three types of parent updates in J :

976 1. A new node (u, i) is inserted at level i of T , and it becomes a center at level $i+1$ of some tree

⁴We choose a random number 10 to make our point simpler; in reality, we construct a $6c\frac{\delta}{\varepsilon^i}$ -net where c is the constant in Definition 8.

977 $J \in \mathcal{J}$. We then have to consider whether to merge some nodes (v, i) in T , which are currently
 978 unmerged in J , to (u, i) in J . This induces parent updates of children of $\psi_J(v, i)$. (A similar but
 979 much easier case is when (u, i) is inserted as a non-center node in a tree J , which we will discuss
 980 more below.)

981 2. A new node (x, i) in T is inserted to split a jump from (x, h) down to (x, l) where $l < i < h$. In
 982 this case, we also split the jump from $\psi_J(x, h)$ to $\psi_J(x, l)$ in T by inserting $\psi_J(x, i)$ in J . Now the
 983 parent of $\psi_J(x, l)$ changes from $\psi_J(x, h)$ to $\psi_J(x, i)$.
 984 3. If (u, i) is created by promoting $(u, i-1)$ in T , then $(u, i-1)$ has to change its parent from an
 985 existing node (v, i) to (u, i) , which induces a parent update of $\psi_J(u, i-1)$.

986 The only non-trivial parent update that we allow in a tree $J \in \mathcal{J}$ is in the edge subdivision. Indeed,
 987 case 2 above is an edge subdivision in J . A subtle point here is that the new subdividing node (x, i)
 988 might possibly be merged to some center node at level i of T . Our idea is to show that the jump isolation
 989 property forbids this case.

990 For case 3, the ideal situation described above happens: by the shifting property, there exists a tree
 991 J such that (v, i) is a center in J and $\psi_J(u, i-1)$ is a child of $\psi_J(v, i)$. This means even if we create a
 992 new node (u, i) in T , then one can show that $d_X(v, u)$ is small and hence (u, i) will be merged to (v, i) .
 993 However, as the (only) child $(u, i-1)$ of (u, i) is already a child of (v, i) in J , we do not have to do
 994 anything in J . For every other tree $J' \in \mathcal{J}$, we still need to insert (u, i) as a new leaf node (and then we
 995 insert a null leaf to be a child of (u, i) in J), but we do not have to move $(u, i-1)$ to be a child of (u, i) .
 996 As a result, we will lose the hierarchical property in trees of \mathcal{J} : a node at level i might not be associated
 997 with points from its children. Fortunately, this property is not important for our end goal, which is to
 998 construct an LSO.

999 Lastly, to handle case 1, we introduce a new rule called *merging through time*. Specifically, we allow
 1000 merging (v, i) into (u, i) in J if (v, i) is added to T after (u, i) . Then case 1 does not happen as (v, i) was
 1001 inserted before (u, i) . When (u, i) is inserted as a non-center node in a tree J , we have to merge (u, i) to
 1002 an existing center node, say (x, i) , in J . But this is an easy case since we only insert leaves to J , so (u, i)
 1003 has no children at the time of its insertion, hence inducing no parent updates.

1004 4.1.1 Dynamic net tree cover construction

1005 First, we color nodes of the dynamic net tree T when nodes are inserted to T . Note that we do not
 1006 remove nodes out of T ; we only mark leaves as deleted, and the same holds for trees in \mathcal{J} . We say that
 1007 two net points (x, i) and (y, i) at level i are *r-close* if $d_X(x, y) < r \frac{\delta}{\epsilon^i}$. Let $\Delta_i = \frac{6c\delta}{\epsilon^i}$ be the *coloring distance*
 1008 at level i . Let k be the maximum number of net points Y_i in a ball of diameter Δ_i . By packing property
 1009 of T and packing property of (X, d_X) , $k = O_\lambda(1)$. Basically, we will assign a color for (u, i) from $[1, 2k]$
 1010 depending on its levels and neighbors; the reason for this was already explained above.

COLORINGNODE(u, i, T):

1. If i is odd, assign (u, i) a smallest color in $[1, k]$ such that the color of (u, i) is different from
 the colors of all level- i nodes within radius Δ_i of u .
2. If i is even, assign (u, i) a smallest color in $[k+1, 2k]$ such that the color of (u, i) is different
 from the colors of all level- i nodes within radius Δ_i of u .

Let $\kappa(u, i)$ be the color of (u, i) . By Theorem 10, all the nodes within distance Δ_i of (u, i) can be
 found in $O_\lambda(1)$ time.

1011 The color of a node (u, i) at level i of T will tell us which tree in \mathcal{J} that (u, i) will be a center (at level
 1012 $j + 1$). Therefore, the number of trees in \mathcal{J} is the number of colors, which is $2k = O_\lambda(1)$, and each tree
 1013 $J \in \mathcal{J}$ will have a *color* $\kappa(J)$ in $[1, 2k]$.

1014 Recall in Definition 6, whenever a point p is inserted to S , the dynamic net tree data structure will
 1015 return a list L_p of $O_\lambda(1)$ nodes that are either changed or inserted to T . We will take this list L_p and call
 1016 $\text{INSERT}(L_p, J)$ in Figure 6 to update every tree $J \in \mathcal{J}$. By the stability property of J , we are only allowed
 1017 to use three following operations as a black box:

- 1018 • $\text{INSERTLEAF}(x, i, J)$: inserting a leaf associated with point x at level i of J as a leaf. One should
 1019 think of (x, i) as a node at level i in J without any children. (If $x = \text{NULL}$, then we call the leaf
 1020 a null leaf.) To keep the pseudocode clean, we do not specify the parent of the new leaf (x, i) .
 1021 The parent will either be clear from the context or if the insertion of (x, i) to J is triggered by the
 1022 insertion of the corresponding node $(x, i - 1)$ to T whose parent is (u, i) , then the parent in J of
 1023 (x, i) is $\psi_J(u, i)$.
- 1024 • $\text{SUBDIVIDING}(u, v, z, J)$: subdividing an edge between u (parent) and v (child) in J by creating a
 1025 new node z and adding z between u and v .

1026 We will use notation (x, i, T) and (x, i, J) to distinguish a node (x, i) at level i in T and a node (x, i)
 1027 at level i in J , respectively.

1028 Every step in $\text{INSERT}(L_p, J)$ in Figure 6 is self-explained, except step 3(d). Recall that T and J are off
 1029 by one level. It is possible that for a point $x \in S$, its corresponding node $(x, 0)$ in T is merged to some
 1030 other node at level 0 in T , and hence in this case, we will have to create a node $(x, 0)$ at level 0 of J to
 1031 guarantee that level 0 of J contains every (non-deleted) point in S .

1026 $\text{INSERT}(L_p, J)$:

1. Sort nodes in L_p by descending order of levels. Then we consider every node $(x, i, T) \in L_p$ in
 the sorted order and apply steps 2 and 3 below.
2. If (x, i, T) splits a jump from (x, h) to (x, l) , call $\text{SUBDIVIDING}(\psi_J(x, h), \psi_J(x, l), (x, i + 1), J)$.
 Then set $\psi_J(x, i, T) \leftarrow (x, i + 1, J)$.
3. If (x, i, T) is a new-point node or a promoting node:
 - If $\kappa(x, i) = \kappa(J)$, then (x, i) is a center in J . We call $\text{INSERTLEAF}(x, i + 1, J)$ and set
 $\psi_J(x, i, T) \leftarrow (x, i + 1, J)$.
 - Otherwise, we find a center (w, i, T) is $3c$ -close to (x, i, T) in T such that $\kappa(w, i) = \kappa(J)$
 $d_X(w, x) < 3c \frac{\delta}{\epsilon^i}$. There is at most once such node (w, i, T) , since we assign different
colors to different nodes within the distance $\Delta_i = 6c \frac{\delta}{\epsilon^i}$.
 - If (w, i, T) exists, set $\psi_J(x, i, T) \leftarrow \psi_J(w, i, T)$. $\ll \text{merge } (x, i) \text{ to } (w, i) \gg$
 - Otherwise, call $\text{INSERTLEAF}(x, i + 1, J)$ and set $\psi_J(x, i, T) \leftarrow (x, i + 1, J)$.
 - If (x, i, T) is a promoting node in $\text{PROMOTE}(x, i - 1, T)$, call $\text{INSERTLEAF}(\text{NULL}, 0, J)$ to
create a null leaf and assign the null leaf as a child of $\psi_J(x, i, T)$ in J .
 - If (x, i, T) is a new-point node and $i = 0$, call $\text{INSERTLEAF}(x, 0, J)$ to create $(x, 0, J)$ as a
child of $\psi_J(x, 0, T)$.

Figure 6: Updating J when a new node is inserted to T .

1032 **Remark 2.** If we update J following the algorithm in Figure 6, the top of a jump in J can have two or
 1033 more children, this occurs when we merge (x, i, T) to (w, i, T) , and any of them is the top of a jump in
 1034 T . In the dynamic pairing algorithm in Section 4.2, it would be much easier if the top of a jump has
 1035 only one child. To guarantee this, whenever we have a jump from (x, i, T) down to (x, j, T) (at step 2,
 1036 3(c) or 3(d) of INSERT), we create (x, i, J) as a child of $\psi_J(x, i)$ and make a jump from (x, i, J) down to
 1037 $\psi_J(x, j) = (x, j + 1, J)$. Since the jump starting at (x, i, T) is $3c$ -isolated, $(x, i - 1, T)$ is not $3c$ -close with
 1038 any node at the same level. Thus, (x, i, J) has only one child $\psi_J(x, j) = (x, j + 1, J)$. Now, the top of any
 1039 jump in J has only one child, and for any node (u, i, J) has two or more children, (u, i, J) and all of its
 1040 children are non-hidden nodes.

1041 4.1.2 Analysis

1042 In this section, we show all the properties of the dynamic net tree cover stated in Definition 8. We start
 1043 with the partial isomorphism property.

1044 **Lemma 5.** Every $J \in \mathcal{J}$ is a tree and satisfies partial isomorphism property: for any node $(x, i + 1, J)$ in
 1045 J where $i \geq 0$, if (x, i, T) does not have parent update except by edge subdividing, then $(x, i + 1, J)$ is a
 1046 child of $\psi_J(u, i')$ where (u, i', T) is the parent of (x, i, T) in T . Furthermore, if (x, i, T) is a promoted
 1047 node whose parent before the promotion is $(v, i + 1, T)$, then $(x, i + 1, J)$ is a child of $\psi_J(v, i + 1)$.

1048 **Proof:** Observe that we only modify J by creating leaves or subdividing edges, and hence J is a tree.
 1049 Inductively, assume that J satisfies the partial isomorphism property before an update, and we have to
 1050 show that it holds after an update, and specifically, if (x, i, T) has no parent update, then the partial
 1051 isomorphism holds for (x, i, T) .

1052 Suppose that (x, i, T) has a parent update by splitting a jump from (x, j, T) to (x, i, T) for $i < j$, and
 1053 in this case, a node (x, k, T) is inserted between them where $i < k < j$. Now the parent of (x, i, T) is
 1054 (x, k, T) . By induction, in J , there is a corresponding jump from $\psi_J(x, j)$ down to $\psi_J(x, i)$. Then in
 1055 step 2 of the insert algorithm, we add $\psi_J(x, k)$ to split this jump, and hence the parent of (x, k, T) is
 1056 $\psi_J(x, k)$, giving the partial isomorphism property.

1057 The last claim that $(x, i + 1, J)$ is a child of $\psi_J(v, i + 1)$ follows from step 3(a). \square

1058 Next, we show the packing and covering properties.

1059 **Lemma 6.** For any $J \in \mathcal{J}$, points at level $i + 1$ is an $O(\frac{\delta}{\epsilon^i})$ -net of S :

- 1060 • [Packing.] For any pair of nodes $(u, i + 1, J)$ and $(v, i + 1, J)$, $d_X(u, v) > \frac{1}{4} \frac{\delta}{\epsilon^i}$.
- 1061 • [Covering.] For any child (x, i, J) of $(v, i + 1, J)$, $d_X(x, v) < 4c \frac{\delta}{\epsilon^i}$. This means, for any descendant
 1062 (y, j + 1, J) of $(v, i + 1, J)$ for $j \leq i$, $d_X(v, y) < 5c \frac{\delta}{\epsilon^i}$.

1063 **Proof:** Since nodes at level $i + 1$ in J is a subset of nodes at level i in T , for any $(u, i + 1, J)$ and $(v, i + 1, J)$,
 1064 the packing property of J follows from that of T (see also Theorem 10).

1065 For covering property of J , first we observe that there exist $(x, i - 1, T)$ and (v, i, T) in T corresponding
 1066 to (x, i, J) and $(v, i + 1, J)$, respectively. If (v, i, T) is the parent of $(x, i - 1, T)$, then by Theorem 10,
 1067 $d_X(x, v) \leq \phi \frac{\delta}{\epsilon^i} \leq 4c \frac{\delta}{\epsilon^i}$ as $c \geq 1$. However, it is possible that (v, i, T) is not the parent of $(x, i - 1, T)$ since
 1068 the parent of $(x, i - 1, T)$, denoted by (u, i, T) , is merged to (v, i, T) . In this case, by the construction in
 1069 step 3b(i) of INSERT(L_p, J), $d_X(u, v) < 3c \frac{\delta}{\epsilon^i}$. Since (u, i) is the parent of $(x, i - 1)$ in T , by the covering
 1070 property of T , $d_X(x, u) \leq \phi \frac{\delta}{\epsilon^i}$. By triangle inequality:

$$1071 \begin{aligned} d_X(x, v) &\leq d_X(x, u) + d_X(u, v) \\ &< (\phi + 3c) \frac{\delta}{\epsilon^i} \leq 4c \frac{\delta}{\epsilon^i} \quad (\text{since } \phi < 1 \text{ and } c \geq 4) \end{aligned}$$

Finally, we bound the distance from $(v, i+1, J)$ to its descendants. By induction, the distance from any child (x, i, J) of $(v, i+1, J)$ to one its descendants, say $(y, j+1, J)$ is at most $5c \frac{\delta}{\varepsilon^{i-1}}$. By the covering of J , $d_X(v, x) < 4c \frac{\delta}{\varepsilon^i}$. By triangle inequality:

$$\begin{aligned} d_X(v, y) &\leq d_X(v, x) + d_X(x, y) \\ &< (4c + 5c\varepsilon) \frac{\delta}{\varepsilon^i} \leq 5c \frac{\delta}{\varepsilon^i} \quad (\text{since } \varepsilon \leq 1/20 \text{ in Definition 8}) \end{aligned}$$

as desired. \square

Next, we show the shifting property of \mathcal{J} . Recall that we set the isolation parameter $b = 3c$ for the dynamic net tree T so that every jump is $3c$ -isolated. We will use the jump isolation property extensively to show the shifting property. First, we claim that:

Claim 1. *If (u, i) and (v, i) are $3c$ -close, there exists a tree $J \in \mathcal{J}$ such that $\psi_J(u, i) = \psi_J(v, i)$. Furthermore, if we add (u, i, T) to T before (v, i, T) then J and (u, i, T) have the same color.*

Proof: W.l.o.g, assume that we add (u, i, T) to T , either explicitly as a node at level i or as a hidden node in a jump, before (v, i, T) . If (u, i, T) is a hidden node in a jump, since every jump is $3c$ -isolated, $d_X(u, y) \geq 3c \frac{\delta}{\varepsilon^i}$ for any node (y, i) at level i ; this contradicts that (u, i) and (v, i) are $3c$ -close. Thus, (u, i, T) is not a hidden node, and therefore, there exists a tree $J \in \mathcal{J}$ where $\kappa(J) = \kappa(u, i)$.

When (u, i, T) is added to T , the insert procedure will add $(u, i+1, J)$ as a node in J and assign $\psi_J(u, i) = (u, i+1, J)$. Since $d_X(u, v) < 3c \frac{\delta}{\varepsilon^i} < \Delta_i$, we have $\kappa(v, i) \neq \kappa(u, i)$. It follows that $\kappa(v, i) \neq \kappa(J)$. Thus, when (v, i) was inserted to T , (v, i, T) and (u, i, T) satisfy the step 3(b) of $\text{INSERT}(L_p, J)$ with $w = u$ and $x = v$, and therefore, step i in 3(b) will set $\psi_J(v, i) \leftarrow \psi_J(u, i)$. \square

Lemma 7. *\mathcal{J} satisfies the shifting property: For every pair of nodes (x, i) and (y, i) with $d_X(x, y) < c \frac{\delta}{\varepsilon^{i+1}}$, there exists a tree $J \in \mathcal{J}$, such that $\psi_J(x, i) = (x, i+1, J)$, $\psi_J(y, i) = (y, i+1, J)$ and they have the same parent in J . Furthermore, let $(u, i+1)$ and $(v, i+1)$ be the parent of (x, i) and (y, i) in T . If we add $(u, i+1)$ before $(v, i+1)$ to T , then $\kappa(J) = \kappa(u, i+1)$, and $\psi_J(u, i+1)$ is the parent of both $\psi_J(x, i)$ and $\psi_J(y, i)$.*

Proof: We consider a pair of net points $x, y \in Y_i$ whose $d_X(x, y) < c \frac{\delta}{\varepsilon^{i+1}}$. If $(x, i+1)$ (or $(y, i+1)$) is a hidden node, then the $3c$ -jump isolation property is violated at $(x, i+1)$ (or $(y, i+1)$ respectively). Thus, the parents of (x, i) and (y, i) must be non-hidden nodes at level $i+1$ in T . We consider two cases:

If (x, i, T) and (y, i, T) have the same parent in T , denoted by $(u, i+1, T)$. Let J be a tree in \mathcal{J} whose color $\kappa(J) = \kappa(u, i+1)$. Since the sets of colors used for any two consecutive levels of J are disjoint, no node (w, i) at level i of T has $\kappa(w, i) = \kappa(J)$. Therefore, when (x, i) and (y, i) were inserted to J , step 3(b-ii) will be executed: two nodes $(x, i+1, J)$ and $(y, i+1, J)$ will be created as children of $\psi_J(u, i+1)$, and we set $\psi_J(x, i) \leftarrow (x, i+1, J)$ and $\psi_J(y, i) \leftarrow (y, i+1, J)$. Thus $\psi_J(x, i)$ and $\psi_J(y, i)$ have the same parent.

Otherwise, (x, i, T) and (y, i, T) have different parents in T , denoted by $(u, i+1, T)$ and $(v, i+1, T)$ respectively. By covering property of T , $d_X(u, x)$ and $d_X(v, y)$ are both at most $\phi \frac{\delta}{\varepsilon^{i+1}}$. Since $d_X(x, y) < c \frac{\delta}{\varepsilon^{i+1}}$, and by triangle inequality, we obtain:

$$\begin{aligned} d_X(u, v) &\leq d_X(u, x) + d_X(x, y) + d_X(y, v) \\ &< \phi \frac{\delta}{\varepsilon^{i+1}} + c \frac{\delta}{\varepsilon^{i+1}} + \phi \frac{\delta}{\varepsilon^{i+1}} \\ &< 3c \frac{\delta}{\varepsilon^{i+1}} \quad (\text{since } \phi < 1 \leq c) \end{aligned} \tag{4}$$

1108 Suppose w.l.o.g that we add $(u, i+1, T)$ to T before $(v, i+1, T)$. Let J be the tree in \mathcal{J} such that
1109 $\kappa(J) = \kappa(u, i+1)$. By Claim 1, $\psi_J(u, i+1) = \psi_J(v, i+1)$. By the same argument in the first case, since
1110 the sets of colors between two consecutive levels are disjoint, two nodes $(x, i+1, J)$ and $(y, i+1, J)$ will
1111 be created as children of $\psi_J(u, i+1)$ and $\psi_J(v, i+1)$, respectively. Since $\psi_J(u, i+1) = \psi_J(v, i+1)$, we
1112 conclude that $\psi_J(x, i)$ and $\psi_J(y, i)$ have the same parent. \square

1113 Finally, we show the pairwise covering property of \mathcal{J} .

1114 **Lemma 8.** *\mathcal{J} satisfies pairwise covering: For every two points $x_0, y_0 \in S$ such that $d_X(x_0, y_0) \in (\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i}]$,
1115 there exists a tree J such that $(x_0, 0, J)$ and $(y_0, 0, J)$ have the same ancestor at level $i+1$ of J .*

1116 **Proof:** Recall that we color nodes in T by the parity of levels. Let $\mathcal{J}_0 \subseteq \mathcal{H}$ be the subset of trees whose
1117 centers are colored at level i ; these colored nodes will appear in level $i+1$ of trees in \mathcal{J}_0 . Since all nodes
1118 at level $i-1$ in T have different colors with $\kappa(J_t)$, Step 3(b-i) in $\text{INSERT}(L_p, J)$ will not be applicable to
1119 trees in \mathcal{J}_0 when considering nodes at level $i-1$ in L_p . Therefore, step 3(b-ii) will be executed, which
1120 means, for every $J \in \mathcal{J}_0$ and every $(p, i-1, T)$, step 3(b-ii) $\psi_J(p, i-1) \leftarrow (p, i, J)$.

1121 Observe that in different trees $J \in \mathcal{J}_0$, we may have different ancestors at level i of the leaf node
1122 $(x_0, 0, J)$. For each $J_t \in \mathcal{J}_0$, let $(x_t, i-1, T)$ and $(y_t, i-1, T)$ be nodes in T such that (x_t, i, J_t) and (y_t, i, J_t)
1123 are respectively the ancestor at level i of $(x_0, 0, J_t)$ and $(y_0, 0, J_t)$. Let $R = \{x_1, x_2, \dots, x_{|\mathcal{J}_0|}, y_1, y_2, \dots, y_{|\mathcal{J}_0|}\}$;
1124 R contains net points at level $i-1$ of T . Now, we show that every two points in R have distance at most
1125 $c \frac{\delta}{\varepsilon^i}$. By triangle inequality, for any $t, t' \in \{1, \dots, |\mathcal{J}_0|\}$, we have:

$$\begin{aligned} d_X(x_t, y_{t'}) &\leq d_X(x_t, x_0) + d_X(x_0, y_0) + d_X(y_0, y_{t'}) \\ d_X(x_t, x_{t'}) &\leq d_X(x_t, x_0) + d_X(x_0, x_{t'}) \\ d_X(y_t, y_{t'}) &\leq d_X(y_t, y_0) + d_X(y_0, y_{t'}) \end{aligned} \tag{5}$$

1127 Since $(x_0, 0, J_t), (y_0, 0, J_t)$ are descendants of (x_t, i, J_t) and $(y_{t'}, i, J_{t'})$, by Lemma 6, $d_X(x_t, x_0)$ and
1128 $d_X(y_{t'}, y_0)$ are bounded by $5c \frac{\delta}{\varepsilon^{i-1}}$. By the assumption of the lemma, $d_X(x_0, y_0) < \frac{2\delta}{\varepsilon^i}$. Plugging these
1129 bounds to Equation (5), we have:

$$\begin{aligned} d_X(x_t, y_{t'}) &< 5c \frac{\delta}{\varepsilon^{i-1}} + \frac{2\delta}{\varepsilon^i} + 5c \frac{\delta}{\varepsilon^{i-1}} \\ &= (2 + 10c\varepsilon) \frac{\delta}{\varepsilon^i} \\ &\leq c \frac{\delta}{\varepsilon^i} \quad (\text{since } \varepsilon \leq \frac{1}{20} \text{ and } c \geq 4 \text{ by Definition 8}) \end{aligned}$$

1131 Since a jump is $3c$ -isolated, every node $(x, i-1, T)$ corresponding to a point $x \in R$ has the (non-
1132 hidden) parent at level i .

1133 Let (s, i, T) be the parent of a node in R that is added first to T among all the parents of all the
1134 nodes in R . Let $J \in \mathcal{J}_0$ be the tree such that $\kappa(J) = \kappa(s, i)$. By Lemma 7, $\psi_J(s, i)$ is the parent of
1135 $\psi_J(x, i-1) = (x, i, J)$ for every $x \in R$. Since $J \in \mathcal{J}_0$, there exist $(x', i-1, T)$ and $(y', i-1, T)$ where
1136 $x', y' \in R$ that are ancestors of $(x_0, 0, J)$ and $(y_0, 0, J)$, respectively, such that $\psi_J(x', i-1) = (x', i, J)$
1137 and $\psi_J(y', i-1) = (y', i, J)$. Since $\psi_J(x', i-1)$ and $\psi_J(y', i-1)$ have the same parent, which is $\psi_J(s, i)$,
1138 the lemma holds. \square

1139 **Proof (Proof of Theorem 11):** By Lemma 5 and Lemma 6, we show that every $J \in \mathcal{J}$ satisfies packing
1140 property, covering property, and is partial isomorphic with T . By Lemma 7, \mathcal{J} satisfies shifting property,
1141 and by Lemma 8, \mathcal{J} satisfies the covering property. Since \mathcal{J} has $2k$ trees where $k = O_\lambda(1)$, $|\mathcal{J}| \in O_\lambda(1)$.

1142 Observe that in $\text{INSERT}(L_p, J)$, we only call edge subdivisions and insert leaves to J and hence every
 1143 tree in J is stable. Furthermore, as $|L_p| = O_\lambda(1)$, we only call $O_\lambda(1)$ update operations to J . Coloring
 1144 a node in T takes $O(1)$ time as noted in the procedure. The most expensive step (per node in L_p) in
 1145 $\text{INSERT}(L_p, J)$ is to find $3c$ -close nodes for (x, i, T) in step 3(b), which can also be done in $O_\lambda(1)$ time by
 1146 Theorem 10. Therefore, the total running time overhead is $O_\lambda(1)$ as claimed. \square

1147 4.2 Dynamic Pairing

1148 In Section 3, we showed the static construction for pairwise tree cover from a net tree. Here we adapt the
 1149 static construction to construct a dynamic pairwise tree cover from a dynamic net tree cover as claimed
 1150 in Theorem 12, which we restate below.

1151 **Theorem 12 (Dynamic Pairing).** *Let \mathcal{J} be a dynamic stable (δ, ε) -net tree cover constructed from a
 1152 (δ, ε) -net tree in Theorem 11. Then we can construct from \mathcal{J} a collection of **stable** dynamic PITs \mathcal{T} such
 1153 that (i) $|\mathcal{T}| = \varepsilon^{-O(\lambda)}$ and (ii) for every points $x, y \in S$ where $d_X(x, y) \in [\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i}]$, there exists a PIT $T' \in \mathcal{T}$
 1154 such that a node at level i of T' is $O(\varepsilon)$ -close to (x, y) . Furthermore, every update to a tree in \mathcal{J} can be
 1155 translated into $\varepsilon^{-O(\lambda)}$ updates to \mathcal{T} that can be identified in $\varepsilon^{-O(\lambda)}$ time.*

1156 For each stable (δ, ε) -net tree $J_g \in \mathcal{J}$, we will construct a collection of PITs \mathcal{T}_g ; the final set of PITs
 1157 contains all PITs $\mathcal{T} = \mathcal{T}_1 \cup \dots \cup \mathcal{T}_{|\mathcal{J}|}$. We guarantee that there exists a PIT $T \in \mathcal{T}_g$ such that T contains
 1158 (x, y, i) ⁵ pairing up (x, i) and (y, i) if two following conditions hold:

- 1159 (a) $d_X(x, y) \in R_i$ where $R_i = \left((1 - 5c\varepsilon)\frac{\delta}{\varepsilon^i}, (2 + 5c\varepsilon)\frac{\delta}{\varepsilon^i}\right]$, where c is the same constant that we use in
 1160 net tree cover.
- 1161 (b) (x, i) and (y, i) have the same parent in J_g .

1162 Note that in the static construction, for net points x, y at level $i - 1$ of the (static) net tree, we say
 1163 (x, y) is a blue edge if $d_X(x, y) \in R_i$, a red edge if $d_X(x, y) \leq s_i$. Here, we do not use red edges, and
 1164 instead use pairs (x, y) if (x, i) and (y, i) have the same parent in one of the stable net trees. Recall that
 1165 in the static construction, children of a pairwise node (x, y, i) include the corresponding pairwise nodes
 1166 of $(x, i - 1)$'s children, $(y, i - 1)$'s children, and children of some unmatched node $(z, i - 1)$. Here, if
 1167 (z, i) is added to J_g before (x, i) and (y, i) , then we have to change the parent for the corresponding
 1168 pairwise node of (z, i) 's children, from (z, z, i) to (x, y, i) . As parent updates make PITs unstable, we
 1169 have to avoid this case. Specifically, in the dynamic pairing algorithm, we relax the static algorithm
 1170 in that the children of (x, y, i) include the corresponding pairwise node of (x, i) 's children and (y, i) 's
 1171 children only. Intuitively, (x, y, i) is created as merging two subtrees of J_g rooted at (x, i) and (y, i) ,
 1172 and (x, y, i) becomes an $O(\varepsilon)$ -node for any pair (x_0, y_0) where x_0 and y_0 are respectively a point in
 1173 descendants of (x, i) and (y, i) in J_g .

1174 In more detail, for some $J_g \in \mathcal{J}$ where (x, i) and (y, i) have the same parent, suppose that (y, i)
 1175 is added to J_g after (x, i) . We visit all PITs in \mathcal{T}_g to find a tree T containing (x, x, i) ; we will show
 1176 that such tree T exists as long as \mathcal{T}_g has sufficiently (but still $\varepsilon^{-O(\lambda)}$) many trees. Then we rename the
 1177 corresponding pairwise node in T of (x, i) from (x, x, i) to (x, y, i) . By applying the pairwise covering
 1178 property of \mathcal{J} , we guarantee that every pair of points in S with a certain range of distance has an
 1179 $O(\varepsilon)$ -close node. We will show that PITs in \mathcal{T}_g have the same types of updates as J_g , and since J_g is
 1180 stable, every PIT is stable.

1181 Note that we do not use jump terminology in dynamic PITs. PITs still have a “long” edge between a
 1182 node at level j and a node at level k where $j > k + 1$; this long edge corresponds to some jump in the

⁵In the static construction, the level of a PIT and the level of a net tree differ by 1; here the levels of a stable net tree and PITs derived from it are the same.

1183 corresponding stable net tree. Therefore, splitting a jump in a net tree will correspond to subdividing a
 1184 (long) edge in a PIT.

1185 Now, we describe our dynamic pairing algorithm.

1186 **Dynamic pairing algorithm.** If a leaf $(q, 0)$ is marked as deleted in J_g , in every PIT of \mathcal{T}_g , we mark its
 1187 corresponding pairwise nodes $(q, q, 0)$ as deleted. When J_g has a new node (p, i) , we create pairwise
 1188 nodes for (p, i) in PITs of \mathcal{T}_g depending on the type of (p, i) :

- 1189 (1) If (p, i) splits a jump from (p, j) down to (p, k) in J_g where $j > i > k$: since (p, j) has only one child,
 1190 (p, i) is not paired up with any node at the same level. Thus, for every PIT $T \in \mathcal{T}_g$, let (p_1, p_2, j)
 1191 and (p, p, k) be the corresponding pairwise node in T of (p, j) and (p, k) , respectively—note that
 1192 (p, k) is the only child of (p, j) before adding (p, i) , thus its corresponding pairwise node is single
 1193 label. We create (p, p, i) by subdividing the edge from (p_1, p_2, j) down to (p, p, k) .
- 1194 (2) If $i > 0$ and (p, i) is a child of $(q, i + 1)$: if there is some node (x, i) where $d_X(p, x) \in R_i$ and (x, i)
 1195 is also a child of $(q, i + 1)$, we will call the *dynamic matching algorithm* described below. This
 1196 algorithm will create a new pairwise node (x, p, i) in a PIT T of \mathcal{T}_g (as well as several single-label
 1197 nodes in some other trees in \mathcal{T}_g).
- 1198 (3) If $i = 0$, we create a leaf $(p, p, 0)$ in every PIT $T \in \mathcal{T}_g$, note that p can be a null point. To find parent
 1199 for $(p, p, 0)$ in a PIT T , let (p', i') be the parent of $(p, 0)$ in J_g , and (p_1, p_2, i') be the corresponding
 1200 pairwise node in T of (p', i') . We make $(p, p, 0)$ a child of (p_1, p_2, i') .

1201 **Dynamic matching algorithm.** This algorithm applies to the case where a new node (p, i) of a stable
 1202 net tree J_g has (at least one) sibling (x, i) such that $d_X(x, p) \in R_i$. Let $(q, i + 1)$ be the parent of (p, i)
 1203 (and also (x, i)). Let \mathcal{I} be the set of PITs in \mathcal{T}_g that do not have a pairwise node of p ; initially, $\mathcal{I} = \mathcal{T}_g$.
 1204 For every child (x, i) of $(q, i + 1)$ in J_g , if $d_X(x, p) \in R_i$, let T be a PIT in \mathcal{I} where the corresponding
 1205 pairwise node of (x, i) is (x, x, i) ; in the analysis below, we will show that T exists. Then, we create the
 1206 corresponding pairwise node of (p, i) in T by simply renaming the corresponding pairwise node of (x, i)
 1207 from (x, x, i) to (x, p, i) . Finally, for every remaining tree T' in \mathcal{I} , as T' does not have a corresponding
 1208 pairwise node of (p, i) , we create (p, p, i) as a child of the corresponding pairwise node of $(q, i + 1)$ in T' .

1209 **Analysis.** We now analyze the dynamic pairing algorithm. First, we show a bound on $|\mathcal{J}_g|$ for nice
 1210 properties assumed in the dynamic pairing algorithm to exist.

1211 **Lemma 9.** *It suffices to maintain \mathcal{T}_g that has $|\mathcal{T}_g| = \varepsilon^{-O(\lambda)}$ trees. Furthermore, given a new node (p, i)
 1212 in J_g , the dynamic matching algorithm runs in $\varepsilon^{-O(\lambda)}$ time to update \mathcal{T}_g , and guarantees that: for any
 1213 node (x, i) in J_g where (x, i) and (p, i) have the same parent and $d_X(p, x) \in R_i$, there exists a PIT T in
 1214 \mathcal{T}_g that contains (x, p, i) .*

1215 **Proof:** For every child (x, i) of $(q, i + 1)$ where $d_X(x, p) \in R_i$, the algorithm finds a PIT T containing
 1216 (x, x, i) to create (x, p, i) . Here we show that by constructing a sufficiently large (but still $\varepsilon^{-O(\lambda)}$) number
 1217 of PITs in \mathcal{T}_g , such a tree T is guaranteed to exist.

1218 Let X_i be the set of points labeling nodes at level i of J_g . For $x \in X_i$, we define $N_b(x) = \{y \in X_i :
 1219 d_X(x, y) \in R_i\}$. Let δ_b be the maximum size of $N_b(x)$ for every $x \in X_i$. By packing property of J_g , it
 1220 holds that $\delta_b = \varepsilon^{-O(\lambda)}$, since the minimum distance of points in X_i is $\Theta(\frac{\delta}{\varepsilon^{i-1}})$, while $R_i \in \Theta(\frac{\delta}{\varepsilon^i})$. Consider
 1221 a point $x \in N_b(p)$, observe that at most $\delta_b - 1$ points in $N_b(x) \setminus \{p\}$ that can be paired with x . Therefore,
 1222 there are at most $\delta_b - 1$ PITs in \mathcal{T}_g where (x, i) has a double-label pairwise node. If we maintain δ_b
 1223 trees in \mathcal{T}_g , there always exists a PIT T to pair up x and p .

1224 We now analyze the running time per update. By the packing and covering properties of J_g , $(q, i+1)$
 1225 has $\varepsilon^{-O(\lambda)}$ children, and we can check the pairwise node of a $(q, i+1)$'s child in a PIT with $O(1)$ time.
 1226 For a PIT, we create a pairwise node (p, p, i) or rename (x, x, i) to (x, p, i) in $O(1)$ time. Since there are
 1227 $\varepsilon^{-O(\lambda)}$ trees, the dynamic matching algorithm totally runs in $\varepsilon^{-O(\lambda)}$ time. \square

1228 **Lemma 10.** *For every two points $x, y \in S$ where $d_X(x, y) \in (\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i}]$, there exists a PIT $T \in \mathcal{T}$ such that*
 1229 *a node at level i is $O(\varepsilon)$ -close to (x, y) .*

1230 **Proof:** For every two points $x, y \in S$ where $d_X(x, y) \in (\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i}]$, we prove that there exists a PIT that has
 1231 a pairwise node containing both x and y as its descendant leaves. By the pairwise covering property of
 1232 the net tree cover \mathcal{J} , there exists a tree J_g such that $(x, 0)$ and $(y, 0)$ have the same ancestor at level $i+1$ in J_g .
 1233 Let (x', i) and (y', i) be the ancestor at level i in J_g of $(x, 0)$ and $(y, 0)$ respectively, we know that
 1234 (x', i) and (y', i) have the same parent $(u, i+1)$. By the covering property of J_g (Lemma 6), $d_X(x, x')$
 1235 and $d_X(y, y')$ are at most $5c \frac{\delta}{\varepsilon^{i-1}}$. Thus, $d_X(x', y') \in (\frac{\delta}{\varepsilon^i} - 5c \frac{\delta}{\varepsilon^{i-1}}, \frac{2\delta}{\varepsilon^i} + 5c \frac{\delta}{\varepsilon^{i-1}}] = R_i$. Since $d_X(x, y) > 0$,
 1236 we have $x \neq y$. By Remark 2, since $(u, i+1)$ has at least two children, (x', i) and (y', i) are non-hidden
 1237 nodes in J_g . W.l.o.g, suppose that (x', i) is added before (y', i) to J_g . Since (x', i) and (y', i) have the
 1238 same parent in J_g and $d_X(x', y') \in R_i$, when (y', i) is added to J_g , by Lemma 9, there exists $T \in \mathcal{T}_g$ that
 1239 contains (x', y', i) . Observe that we pair up two nodes in J_g only if they have the same parent, thus the
 1240 ancestor at level j in T of $(x, x, 0)$ is the corresponding pairwise node of the ancestor at level j in J_g of
 1241 $(x, 0)$. Therefore, (x, y, i) contains x , and similarly contains y in its descendants.

1242 Now we show that (x', y', i) is $O(\varepsilon)$ -close to the pair (x, y) . Observe that points in descendants of
 1243 (x', y', i) in T is the union of points in descendants of (x', i) and (y', i) in J_g . Thus, for every point t in
 1244 descendants of (x', y', i) in T , $(t, 0)$ must be a descendants of (x', i) or (y', i) in J_g . By the covering of
 1245 J_g (Lemma 6), $d_X(t, \{x', y'\})$, $d_X(x, x')$ and $d_X(y, y')$ are at most $5c \frac{\delta}{\varepsilon^{i-1}}$. Therefore $d_X(t, \{x, y\})$ is at
 1246 most $10c \frac{\delta}{\varepsilon^{i-1}} \leq 10c \varepsilon d_X(x, y)$. \square

1247 **Proof (Proof of Theorem 12):** By Lemma 9, we know that $|\mathcal{T}_g| = \varepsilon^{-O(\lambda)}$, and since $|\mathcal{J}| = O_\lambda(1)$, $\mathcal{T} =$
 1248 $\mathcal{T}_1 \cup \dots \cup \mathcal{T}_{|\mathcal{J}|}$ has totally $\varepsilon^{-O(\lambda)}$ PITs, which proves item (i) of the theorem. By Lemma 10, item (ii) holds.

1249 Since $|\mathcal{T}_g| = \varepsilon^{-O(\lambda)}$, each update of J_g is translated into $\varepsilon^{-O(\lambda)}$ updates of \mathcal{T}_g . For every PIT in \mathcal{T}_g ,
 1250 the dynamic pairing algorithm marks a leaf as deleted with $O(1)$ time, creates (p, p, i) for a new node
 1251 (p, i) in J_g with $O(1)$ time in step 1 and step 3. In step 2, the dynamic matching algorithm updates all
 1252 PITs in \mathcal{T}_g and runs in $\varepsilon^{-O(\lambda)}$ time by Lemma 9. Therefore, the total running time is $\varepsilon^{-O(\lambda)}$.

1253 Now we show that PITs in \mathcal{T}_g are stable. Parent updates occur only in step 1 of the dynamic pairing
 1254 algorithm, where we subdivide an edge. In step 2, the dynamic matching algorithm creates single-label
 1255 pairwise nodes, or renames nodes to create double-label pairwise nodes in PITs. These single-label
 1256 pairwise nodes do not have any child and thus are leaves. In step 3 of the dynamic pairing algorithm,
 1257 we create a leaf in every PIT. Therefore, we update PITs with three types of operations: adding a leaf,
 1258 subdividing an edge, and marking a leaf as deleted; which means PITs are stable. \square

1259 **Remark 3.** *In the construction of the leaf tracker data structure in Section 5, it would be conceptually
 1260 simpler (though technically not needed) if we re-arrange the order of nodes being inserted into PITs and
 1261 guarantee that adding a new leaf always occurs at level 0. Recall that in a tree J_g of net tree cover \mathcal{J} and
 1262 PITs in \mathcal{T}_g , we insert nodes by:*

1. *In J_g , (p, i) splits a jump from (p, j) down to (p, k) . In every PIT of \mathcal{T}_g , we add (p, p, i) by
 subdividing the edge from (p_1, p_2, j) down to (p, p, k) . In this case, we do not add a new leaf in
 any PIT.*
2. *In J_g , we insert a leaf (p, i) at level $i > 0$, after that, we insert a leaf $(p', 0)$, where $p' = p$ or
 $p' = \text{NULL}$.*

1268 (a) If the corresponding pairwise node in a PIT T of (p, i) is (x, p, i) for $x \neq q$, we add $(p', p', 0)$
 1269 as a leaf under (x, p, i) .
 1270 (b) If the corresponding pairwise node of (p, i) in a PIT T is (p, p, i) , it is a new leaf at level $i > 0$
 1271 in T . We need to arrange new nodes of T in this case.
 1272 3. In J_g , we only insert a leaf $(p', 0)$ to the tree, where p' is a new point or $p' = \text{NULL}$ (without adding
 1273 a leaf at level $i > 0$). This case already satisfies that adding a leaf occurs at level 0.

1274 Now we show how to arrange new nodes of PITs in case 2(b). First, we run the dynamic matching
 1275 algorithm in J_g , then: If the corresponding pairwise node in a PIT T of (p, i) is (p, p, i) , let $(q, i+1)$
 1276 be the parent of (p, i) in J_g , and $(q_1, q_2, i+1)$ be the corresponding pairwise node in T of $(q, i+1)$. In
 1277 T , we add $(p', p', 0)$ as a leaf under $(q_1, q_2, i+1)$, then add (p, p, i) by subdividing the edge between
 1278 $(q_1, q_2, i+1)$ and $(p', p', 0)$.

1279 5 Leaf Tracker

1280 In this section, we design the leaf tracker data structure for a dynamic PIT as claimed in Theorem 13 in
 1281 Section 4, which we restate below for convenience.

1282 **Theorem 13.** *We can construct a leaf tracker data structure for maintaining a stable dynamic PIT T with*
 1283 $O(\varepsilon^{-O(\lambda)} + \log(n))$ *time per updating and tracking operation (including INSERTLEAF, SUBDIVIDEEDGE,*
 1284 *DELETELEAF, TRACKLEFTMOSTLEAF, TRACKRIGHTMOSTLEAF). Furthermore, the DFS leaf ordering of T*
 1285 *will be maintained in a doubly linked list, and hence the data structure could support $O(1)$ time per*
 1286 *query (including GETPREDECESSOR, GETSUCCESSOR).*

1287 In this section, all dynamic trees are stable: every update is either inserting a (null or non-null) leaf,
 1288 marking a leaf deleted, or subdivision an edge. Thus, for simplicity, we use the word dynamic tree to
 1289 refer to a stable dynamic tree.

1290 By definition (Definition 10), a leaf tracker has to maintain a DFS ordering of only active leaves.
 1291 Furthermore, GETPREDECESSOR and GETSUCCESSOR operate on σ and have to return active leaves as
 1292 results. Therefore, in maintaining the DFS ordering of the leaves T , we have to skip over inactive leaves
 1293 (which include null and mark-deleted leaves). And this is the key difficult challenge in the design of a
 1294 leaf tracker data structure.

1295 Recall that the DFS leaf ordering σ of T is obtained by visiting the tree and writing down the leaves
 1296 in the DFS order, breaking ties by insertion time. Specifically, children of every node in T are ordered
 1297 linearly by their insertion time. In the DFS order, we prioritize visiting the nodes in T by their insertion
 1298 time: from a node, we visit the older children first. We will use a doubly-linked list to store σ , and hence,
 1299 getting the predecessor and the successor of a point in σ can be done in $O(1)$ time by simply following
 1300 the pointers to the next and previous nodes in the list σ . (Herein, we will slightly abuse the notion by
 1301 using σ to refer to the doubly-linked list representing the DFS ordering σ .)

1302 First, we will handle a simpler case where a dynamic tree (not necessarily a PIT) T only has (non-null)
 1303 leaf insertions and edge subdivisions; there are no marking leaves as deleted or inserting null leaves. We
 1304 will also store the DFS ordering of leaves of T in a doubly-linked list L_T . Our key idea is to construct a
 1305 data structure that could support querying the leftmost and rightmost ⁶ leaves of a given node $u \in T$ in
 1306 $O(\log(n))$ time. When we add a leaf (q, q, k) as a new child of a node u at a level $k > 0$, we query D to
 1307 get the get the rightmost leaf $(x, x, 0)$ of u in $O(\log(n))$ time. Assume that we are in the ideal case where

⁶The left-right order of nodes in T is determined by the insertion time; specifically, earlier inserted nodes are on the left and vice versa.

1308 $(x, x, 0)$ is active. Then we follow the pointer stored at $(x, x, 0)$ to access its position in L_T , and insert q
 1309 after $(x, x, 0)$ in L_T in $O(1)$ time. The time to locate the rightmost leaf, which is $O(\log n)$, dominates the
 1310 total running time to update σ .

1311 To search for a leftmost or rightmost leaf of a node u , the observation is that all the leaves in the
 1312 subtree rooted at a node u form a contiguous subsequence of L_T , where the leftmost (rightmost) leaf is
 1313 the leftmost (rightmost) element of the subsequence. Then, to search for these extreme points of the
 1314 subsequence, we will build a skip list on top of L_T to perform some kind of binary search. However,
 1315 there seems to be no obvious way to assign keys to elements in L_T to construct the skip list. Nodes in
 1316 L_T are not sorted in increasing orders of insertion time, and there is no natural linear order between
 1317 the names of the nodes to use as keys. To solve this problem, we introduce *ancestral arrays* and a data
 1318 structure for maintaining them. Roughly speaking, an *ancestral array* of a node $u \in T$ is an array $O(\log n)$
 1319 “important” ancestors stemming from a centroid decomposition of T (see Definition 11). We will use
 1320 ancestral arrays as “keys” to the skip list. Though there is no linear order between the ancestral arrays to
 1321 use them as keys in the traditional sense, we could use them to determine if a leaf x is a descendant of
 1322 a query node u or not by Lemma 14, which turns out to be sufficient for binary search using skip lists.
 1323 There are several subtleties in the implementation, which we will discuss in detail later in Section 5.1.
 1324 Our ultimate result is the following data structure.

1325 **Lemma 11.** *Let T be a dynamic rooted tree of n nodes under updates by adding new leaves and
 1326 subdividing edges. Then, we can construct a data structure with $O(n)$ space that maintains the DFS leaf
 1327 ordering of T in a doubly-linked list with $O(\log(n))$ time per update and tracking operation (including
 1328 `INSERTLEAF`, `SUBDIVIDEEDGE`, `TRACKLEFTMOSTLEAF`, `TRACKRIGHTMOSTLEAF`).*

1329 Next, we design a data structure for a PIT which could contain inactive nodes. There are two key
 1330 challenges: (a) a node u might be inactive before the insertion of a new active leaf $(q, q, 0)$ and hence,
 1331 no descendant leaves of u will appear in the DFS leaf ordering σ since they are also inactive; (b) if u is
 1332 active, it is possible that most of its descendant leaves are inactive, including its leftmost and rightmost
 1333 descendant leaves. (Recall that a node in T is active if it has at least one active descendant leaf, and
 1334 inactive otherwise.) To resolve these issues, our basic idea is that, given a PIT T , if we iteratively contract
 1335 every inactive node to its parent until there is no more inactive node, then the resulting tree only has
 1336 active nodes and hence we could apply the data structure, denoted by D , for trees without inactive
 1337 nodes. Of course, we will not explicitly contract inactive nodes, as if we do so, when a new active leaf is
 1338 inserted as a child of an inactive node, it could trigger a large number of nodes to change their status
 1339 from inactive to active, resulting in a large amount of time to undo the contractions. Instead, we design
 1340 a new data structure called active tracker (see Section 5.2.2) that supports two important operations: (i)
 1341 given a node u in a PIT, returns an active descendant leaf of u , if any, and (ii) given an inactive node u ,
 1342 returns the lowest active ancestor of u . Operation (i) allows us to access an active leaf of a node to start
 1343 the binary search on σ using skip lists stored in D . Operation (ii) provides a kind of implicit contraction:
 1344 if a new active leaf $(q, q, 0)$ is inserted as a child of an inactive leaf u , we could conceptually think of
 1345 $(q, q, 0)$ as a new child of the lowest active ancestor v of u in the contracted tree, and hence we could
 1346 call an update to D to insert a new child to v . A subtle point is that the new leaf $(q, q, 0)$ might not be
 1347 the rightmost leaf in the DFS ordering of descendant leaves of v , since the (inactive) child of v that is an
 1348 ancestor of u , denoted by x , might have insertion time smaller than other children of v . In this case, we
 1349 insert $(q, q, 0)$ next to the rightmost leaf of an active child y of v whose insertion time is largest among
 1350 all children of v with insertion time smaller than x . All these ideas lead to the following lemma, whose
 1351 proof will be given in Section 5.2.

1352 **Lemma 12.** Suppose that we are given a data structure in Lemma 11, then we can construct a data
 1353 structure for maintaining the DFS leaf ordering of any given PIT in a doubly-linked list with $O(\varepsilon^{-O(\lambda)} +$
 1354 $\log(n))$ time per update and tracking operation.

1355 Observe that Lemma 11 and Lemma 12 together imply Theorem 13. The rest of this section is
 1356 organized as follows. Section 5.1 construct a data structure for a simpler case as claimed in Lemma 11.
 1357 Section 5.2 shows how to maintain σ and proves Lemma 12.

1358 **5.1 Special Case: Trees without Inactive Nodes**

1359 In this section, we construct a data structure for querying the leftmost and rightmost leaves of a node in
 1360 a dynamic tree T without inactive nodes as claimed in Lemma 11. Note that nodes in T are ordered
 1361 by their insertion times. (Our data structure works for any tree with a linear order between children
 1362 of every node in a tree, not just the linear order by insertion times.) Let L_T be the list of leaves in T
 1363 obtained by visiting T in the DFS order, where children of a node are visited according to their insertion
 1364 times. We observe that:

1365 **Observation 3.** The descendant leaves of any node $u \in T$ form a contiguous subsequence of L_T .

1366 As discussed above, the key idea is to construct a data structure for querying the leftmost and
 1367 rightmost descendant leaves of a node in T . To this end, we need a data structure to maintain an
 1368 ancestral array of every node, each array holds $O(\log n)$ ancestors from a centroid decomposition. We
 1369 say that a path P in T is *monotone* if it is a subpath from a leaf to the root of T .

1370 **Definition 11 (Centroid Decomposition [23] and Ancestral Array).** Given a rooted tree T , a *centroid*
 1371 *path* π of T is a maximal monotone path such that there exists an integer i satisfying $2^i \leq |T(u)| < 2^{i+1}$
 1372 for all node $u \in \pi$, where $|T(u)|$ is the total number of nodes (size) of the subtree rooted at u . We say
 1373 the node at the highest level of π is the *head* of π . A *centroid decomposition* of T is a decomposition into
 1374 a set \mathcal{P} of centroid paths such that every node $u \in T$ has at most $O(\log n)$ centroid paths, each contains
 1375 an ancestor of u , and every ancestor of u (including u) is contained in one of these centroid paths.
 1376 An *ancestral array* of a node u is an array containing the heads of the centroid paths of u .

1377 By the definition of centroid decomposition, the ancestral array of a node has $O(\log n)$ elements, and
 1378 furthermore, the first node of the array is the root of T . We observe that one can extract a data structure
 1379 for maintaining an ancestral array from the data structure for maintaining dynamic weighted ancestors
 1380 in a rooted tree by Kopelowitz and Lewenstein [25]. For completeness, we will review their construction
 1381 and adapt it to our notation by the end of this section.

1382 **Lemma 13 (Kopelowitz and Lewenstein [25], implicit).** Given a rooted tree T of n nodes under up-
 1383 dates by adding new leaves and subdividing edges, there is a data structure that maintains an ancestral
 1384 array for every node in the tree and runs in $O(\log(n))$ time per update.

1385 Let A_v be the ancestral array of a node $v \in T$; note that $A_v[1]$ is the root of T . We now show that
 1386 using ancestral arrays, one can infer if a node x is a leaf descendant of a node u . (We assume that every
 1387 node has a level such that the level of a node is smaller than the level of its parent.)

1388 **Lemma 14.** Let u and x be two nodes in T where x is a leaf. Let h_u be the last element of A_u , and j be
 1389 the index of h_u in A_x . Then x is a descendant leaf of u if and only if either (a) h_u is the last element of
 1390 A_x , or (b) both following conditions hold: (b.1) A_u is a prefix of A_x and (b.2) the parent of $A_x[j+1]$ has
 1391 a level at most the level of u .

1392 **Proof:** Let π be the centroid path containing u . Observe that h_u is the head of π . If $x \in \pi$, then (a)
 1393 holds and x is a descendant leaf of u . On the other hand, if (a) holds, then $x \in \pi$, which means it is a
 1394 descendant leaf of u . It remains to consider the case where $x \notin \pi$. We will show that x is a descendant
 1395 leaf of u if and only if both (b.1) and (b.2) hold.

1396 If u is an ancestor of x , then by definition of ancestral arrays, A_u must be a prefix of A_x ; (b.1) holds.
 1397 Since $x \notin \pi$, j is not the last element of A_x , which means $A_x[j+1]$ exists. Let p be the parent of $A_x[j+1]$,
 1398 observe that $p \in \pi$ and p is the lowest ancestor of x in π . Thus, the level of u must be at least the level
 1399 of p ; (b.2) holds.

1400 On the other hand, assuming that (b.1) and (b.2) hold. By (b.1), we know that π is a centroid path
 1401 of x . As $x \notin \pi$, $A_x[j+1]$ must exist. Let p be the parent of $A_x[j+1]$. By Definition 11, both u, p are in π .
 1402 Since the level of p is at most the level of u by (b.2), u must be an ancestor of p , and therefore, of x . \square

1403 We will construct a skip list structure on top of L_T ; nodes in L_T will be referred to as leaves to be
 1404 distinguished from nodes in the skip list. The skip list has $O(\log n)$ levels to “navigate” L_T ; L_T will be
 1405 at level 0 of the skip list. As we discussed above, the “key” of every element in L_T in the skip list is its
 1406 ancestral array. Though there is no linear order between the ancestral arrays, we could use them to
 1407 determine if a leaf x is a descendant of a query node u or not by Lemma 14. A node in the skip, say
 1408 \tilde{x}_i , at a level i now holds (a) the name of some leaf, say x , in σ as data and (b) three pointers: a **right
 1409 pointer** which points to a node \tilde{y}_i at the same level i where its corresponding leaf y is to the right of x in
 1410 σ , and the **down pointer** which points to \tilde{x}_{i-1} (if any), the node at level $i-1$ holding the same leaf x as
 1411 the data, and the **up pointer** that points to \tilde{x}_{i+1} (if any). Note that each leaf x appears as data in at most
 1412 $O(\log n)$ nodes of the skip list (at different levels)

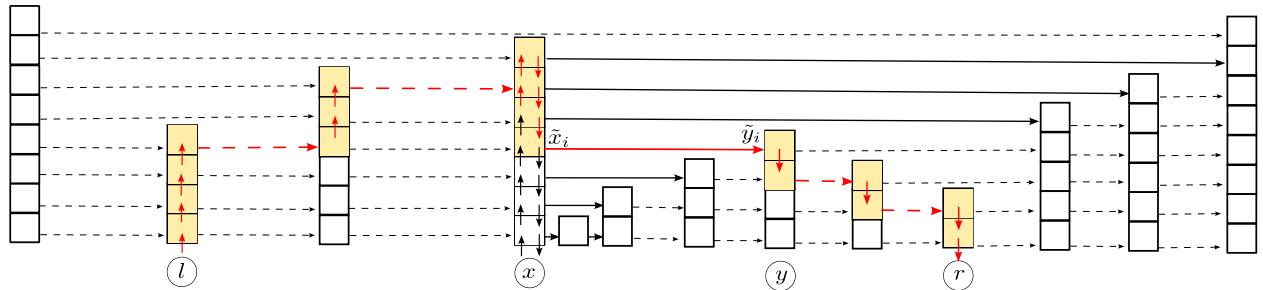


Figure 7: Illustrate the skip list that we build on top of L_T and the process of searching the right-most leaf r of a node u in the tree T . Four circles l, x, y, r are leaves in L_T . A solid arrow is a direct pointer between two nodes. A dashed arrow shows that there is a path between two nodes of different leaves at the same level. We demonstrate direct pointers of nodes \tilde{x}_i for one leaf x , other leaves have the same structure. We start searching at a leaf l of u , then follow pointers (red arrows) of yellow nodes to find r . The going up stage is from l to x and then going down stage is from x to r .

1413 When searching for the rightmost of a node $u \in T$, we take an arbitrary leaf ℓ of u and start the
 1414 search from node $\tilde{\ell}_0$ corresponding to ℓ at level 0 of the skip list, which is L_T ; see the pseudocode in
 1415 Figure 8. There are two stages in the search:

1416 1. **Going up stage.** This stage starts from $\tilde{\ell}_0$, and at an intermediate step, we have a node \tilde{x}_i at level
 1417 i , where its corresponding leaf x is always a descendant of u . We then follow the up pointers to get
 1418 to the node \tilde{x}_{i^*} corresponding to the same leaf x where i^* is the highest level. Then we follow the
 1419 right pointer of \tilde{x}_{i^*} to the next node \tilde{y}_{i^*} , corresponding to a leaf y . If y is a descendant of u ,
 1420 then we will continue this stage by jumping to \tilde{y}_i . Otherwise, we follow the down pointer to \tilde{x}_{i^*-1}
 1421 and start the second stage.

1422 **2. Going down stage.** At an intermediate step, the search is at some node \tilde{x}_i at level i , where its
 1423 corresponding leaf x is always a descendant of u as the first stage. We then follow the right pointer
 1424 to the next node \tilde{y}_{i-1} , corresponding to a leaf y . If y is still a descendant of u , then we jump to \tilde{y}_i .
 1425 Otherwise, we go down one step to \tilde{x}_{i-1} and continue this stage. This stage, and also the search,
 1426 terminates when we reach level 0.

1427 There are two subtle points in implementing the rightmost/leftmost leaf tracker algorithm. The
 1428 minor point is that we will not store the ancestral arrays explicitly in the skip list since their sizes are
 1429 non-constant; instead, we only store pointers to these arrays. The major point is that we have to check
 1430 whether A_u is a prefix of A_x in $O(1)$ time; note that their lengths are $O(\log n)$. We do so by exploiting
 1431 the fact that A_u and A_x are stored as arrays: simply look at the last element, say h_u , of A_u and check if
 1432 $A_u[j] = A_x[j]$ where j is the length of A_u , which is also the index of h_u is A_u .

IsDESCENDANT(u, x, T): *check if a leaf x is a descendant of u*

1. Let j be the last index of A_u , and $h_u \leftarrow A_u[j]$.
2. x is a leaf of u if either one of following conditions holds:
 - (a) The last element of A_x is h_u .
 - (b) If the last element of A_x is not h_u , then both (b.1) and (b.2) hold, where:
 - (b.1) $A_x[j] = h_u$.
 - (b.2) The parent of $A_x[j + 1]$ is at a level at most the level of u .

TRACKRIGHTMOSTLEAF(u, σ, T): *return the rightmost leaf in σ of a given node u*

1. Let ℓ be an arbitrary leaf under the subtree rooted at u .
2. $\text{stage} \leftarrow \text{UP}$, $i \leftarrow 0$ and $\tilde{x}_i = \tilde{\ell}_0$.
3. while $\text{stage} = \text{UP}$ **« up stage »**
 - $\tilde{x}_{i^*} \leftarrow$ the node at highest level corresponding to x .
 - $\tilde{y}_{i^*} \leftarrow \text{RIGHTPOINTER}(\tilde{x}_{i^*})$.
 - if $\text{IsDESCENDANT}(u, y, T) = \text{TRUE}$, $i \leftarrow i^*$ and $\tilde{x}_i \leftarrow \tilde{y}_{i^*}$.
 - otherwise, $\text{stage} \leftarrow \text{DOWN}$, $i \leftarrow i^* - 1$.
4. while $i \geq 0$ **« down stage »**
 - $\tilde{y}_i \leftarrow \text{RIGHTPOINTER}(\tilde{x}_i)$.
 - if $\text{IsDESCENDANT}(u, y, T) = \text{TRUE}$, $\tilde{x}_i \leftarrow \tilde{y}_i$.
 - otherwise, $i \leftarrow i - 1$.
5. Return x .

Figure 8: TRACKRIGHTMOSTLEAF searches the rightmost leaf of u by searching on a skip list.

1433 **Proof (Proof of Lemma 11):** Since all descendant leaves of u form a continuous subsequence of L_T ,
 1434 we can follow pointers of L_T to find the rightmost leaf of u from an arbitrary leaf ℓ . Therefore, the
 1435 correctness of TRACKRIGHTMOSTLEAF follows Lemma 14.

1436 For running time, observe that $\text{ISDESCENDANT}(u, x, T)$ has $O(1)$ running time. Therefore, every
 1437 iteration in the while loops in steps 3 and 4 of $\text{TRACKRIGHTMOSTLEAF}$ run in $O(1)$ time. As the height
 1438 of a skip list is $O(\log n)$, the number of steps going up in the first stage is $O(\log n)$. However, we also
 1439 have to bound the number of steps the search jumps to the right following the right pointers. The
 1440 observation is that, in the first stage, every time we follow the right pointer in the skip list, the distance
 1441 from the starting point of the search to the current node increases exponentially. Thus, the number
 1442 of jumping-to-the-right steps is also $O(\log n)$, implying that the total running time of the first stage is
 1443 $O(\log n)$. The total running time of the second stage is also $O(\log n)$ for the same reason.

1444 Lastly, we need to keep track of L_T once a new leaf is inserted into T . As we do not delete leaves from
 1445 T , when a new leaf x is inserted as children of a node u , it will be the new rightmost descendant leaf of u ,
 1446 and we have to insert x next to its old rightmost descendant leaf, say y . We invoke $\text{TRACKRIGHTMOSTLEAF}$
 1447 to find y and its position in L_T . Since L_T is a doubly linked list, inserting x next to y takes only $O(1)$
 1448 time, making the total time to update L_T $O(\log n)$ per insertion.

1449 As L_T changes, the skip list also has to change. For this purpose, we could use the data structure
 1450 of Munro, Papadakis, and Sedgewick [30] that has only $O(n)$ space. Thus, the total space of our data
 1451 structure is $O(n)$. \square

1452 **Maintaining ancestral arrays** Kopelowitz and Lewenstein [25] studied the dynamic weighted ancestor
 1453 problem, where there is a dynamic weighted tree and the goal is to answer weighted ancestor queries:
 1454 given a node v and a value i , return the first node in the path from v to the root whose value is less than
 1455 i . Note that the weight of a node is higher than that of its parent, and the tree is updated dynamically by
 1456 inserting a leaf or subdividing an edge; there are no deletions. They developed two data structures:

- 1457 1. *Ancestral representative* data structure built on top of the centroid path decomposition (Definition 11) of T . This data structure also maintains for every node v a list of heads of the centroid
 1458 paths of v , called the *head record* of v .
- 1459 2. *Dynamic predecessor* data structure that maintains all the centroid paths in the centroid path
 1460 decomposition and supports predecessor search⁷ on each path.

1462 To query a weighted ancestor, their basic idea was to search the two data structures. First, from the
 1463 head record of v , they determined the (head of the) centroid path π of v that contains the weighted
 1464 ancestor. Then, they executed a *predecessor search* supported by the dynamic predecessor data structure
 1465 to search for the result⁸ on π .

1466 **Remark 4.** *In the work of Kopelowitz and Lewenstein [25], each insertion to the tree induces a constant
 1467 number of what they called predecessor updates. Each predecessor update could be (a) creating at most
 1468 $\log(n)$ nodes in the head record for a new node, (b) adding a new element at the end of the head record,
 1469 (c) changing the value of an element in the head record of a node, and (d) adding a new node into a
 1470 centroid path. The running time of a predecessor update depends on the choice of the data structures
 1471 for the head records and centroid paths. (For our purpose, we simply use an array to store each head
 1472 record and a skip list to store each central path to achieve $O(\log n)$ time per predecessor update.)*

1473 The running time per weighted ancestor query was bounded by the number of predecessor searches.
 1474 One predecessor search is executed on the head record of v , which has size $O(\log n)$, while another
 1475 predecessor search is on a centroid path of size at most n . Thus, the running time of querying a weighted
 1476 ancestor is at most $\max\{T_1(\log(n)), T_2(n)\}$, where T_1 and T_2 depend on the choices of data structures
 1477 for implementing predecessor search on the head records and centroid paths.

⁷In the predecessor search problem, one has to design a data structure for a set of integer keys such that given an integer x , it has to quickly return the largest (smallest) key at most (at least, resp.) x , called the predecessor (successor, resp.) of x .

⁸To be more precise, they either searched on π or the centroid path of v following π .

1478 The result of Kopelowitz and Lewenstein [25] is summarized in the following lemma.

1479 **Lemma 15 (Kopelowitz and Lewenstein [25], Theorem 7.1).** *Given a weighted tree of n nodes that*
1480 *can be updated by adding a new leaf or subdividing an edge, each insertion costs a constant number of*
1481 *predecessor updates, and each weighted ancestor query can be answered by calling a constant number of*
1482 *predecessor searches.*

1483 Here, we do not need to support a weighted ancestor query. Furthermore, we only look for an
1484 $O(\log n)$ search time instead of $O(\log^*(n))$ or faster as in the work of Kopelowitz and Lewenstein [25]).
1485 As a result, we could use much simpler data structures than theirs. Specifically, we will use an array to
1486 store the head record of every vertex; this is our ancestral array. We also use a skip list to store each
1487 centroid path. (Therefore, we can discard most of their dynamic predecessor data structure.) When a
1488 node v is added to the tree, we update its ancestral array in $O(\log n)$ time by looking at the ancestral array
1489 of its parent and add v to an appropriate centroid path also in $O(\log(n))$ time. As noted in Remark 4,
1490 the insertion time is $O(\log(n))$.

1491 For completeness, we now briefly zoom in on the technical ideas of Kopelowitz and Lewenstein [25].
1492 This overview is not necessary to understand our work, and hence, readers could skip this part if needed.

1493 First, as centroid paths are determined based on their sizes, the authors [25] needed to maintain
1494 the sizes of the subtrees rooted at the heads of centroid paths. When a new node is added, a centroid
1495 path π could have a new head: the current head u leaves π and joins to the end of the preceding
1496 centroid path, while the child v of u *in* π becomes the new head of π . And we have to maintain the
1497 size of the subtree, denoted by $|T(v)|$, rooted at v . If v is the only child of u , then $|T(v)| = |T(u)| - 1$.
1498 Otherwise, other child of u are also heads and hence $|T(v)| = |T(u)| - \sum_{v' \in \{\text{children of } u\} \setminus \{v\}} |T(v')| - 1$.
1499 The sum $\sum_{v' \in \{\text{children of } u\} \setminus \{v\}} |T(v')|$ can be maintained directly at u . Therefore, we can compute $|T(v)|$
1500 in constant time. As inserting a node into T could lead to changing at most $\log(n)$ heads, the running
1501 time to maintain the subtree sizes at these heads is $O(\log(n))$.

1502 Maintaining the sizes of the heads is only the first step; the main challenge is to maintain the head
1503 records in an efficient time per insertion. Adding a new node could lead to changing multiple heads
1504 of the centroid paths, which induces updating the head records of many nodes. To solve this problem,
1505 instead of updating the head records of nodes immediately, they waited for more insertions. In the
1506 meantime, a centroid path could be “oversized”, as more nodes are inserted into it but its head is not
1507 updated. They observed that using an “outdated” version of head records still guarantees the correctness
1508 of ancestor queries. Furthermore, this observation allows updating the data structure in the background,
1509 then they could deamortize their construction by recursively splitting the tree into subtrees of $O(\log(n))$
1510 nodes. Ultimately, they achieved a worst-case constant bound on the number of predecessor updates
1511 (and predecessor searches) per insertion (weighted ancestor query, resp.).

1512 5.2 General Case: Maintaining DFS Ordering of a PIT

1513 In this section, we show how to maintain the DFS leaf ordering σ of T as claimed in Lemma 12. We will
1514 store σ as a doubly linked list that only contains *active leaves*. Our goal is to transform a structure D for
1515 trees without inactive leaves, such as the data structure constructed in the previous section, to a data
1516 structure that works for PIT with inactive leaves. We refer readers to the beginning of Section 5 for an
1517 overview of our ideas. A key data structure is an *active tracker* formally defined below.

1518 **Definition 12 (Active Tracker).** *A data structure that maintains a dynamic rooted tree under updates*
1519 *by inserting a leaf, subdividing an edge, and marking a leaf as deleted. It supports three following*
1520 *queries:*

1521 • $\text{ISACTIVE}(u, T)$: check if u is an active node.

- $\text{GETACTIVELEAF}(u, T)$: given a node u , return l_u , where l_u is an active leaf under the subtree rooted at u .
- $\text{GETLOWESTACTIVEANCESTOR}(u, T)$: return the lowest ancestor v of u such that v is an active node.

By the end of this section, we will design an active tracker data structure with linear space that supports fast query time, as claimed in the following lemma. Note that we only apply procedure $\text{GETACTIVELEAF}(u, T)$ on an active node u .

Lemma 16. *There is an active tracker data structure with $O(n)$ space that has $O(\varepsilon^{-O(\lambda)} + \log(n))$ time per update and supports ISACTIVE in $O(1)$ time, and GETACTIVELEAF and $\text{GETLOWESTACTIVEANCESTOR}$ in $O(\log n)$ time.*

Given the active tracker data structure by Lemma 16, we now show how to update the doubly-linked list σ under updates. Conceptually, we could think of σ as a DFS leaf ordering of the tree \hat{T} obtained by iteratively contracting inactive nodes to their parents, and therefore, we could use the data structure in the previous section for maintaining σ . Specifically, a skip list is maintained on top of σ for binary search, as we did with L_T , with ancestral arrays to be keys. However, ancestral arrays of a node in T here could contain inactive nodes. The observation is that in the search of the leftmost/rightmost leaf, we only compare ancestral arrays of active nodes, and by definition, every ancestor of an active node is active. Thus, the ancestral arrays of nodes that we compare during the search only contain active nodes.

5.2.1 Updating σ

When an active leaf $(q, q, 0)$ corresponding to a point $q \in S$ is removed from T , we then follow the pointer at $(q, q, 0)$ to find its corresponding node q in σ . Then, we simply remove q from σ , which can be done in $O(1)$ time since σ is a doubly-linked list. It remains to consider insertions. If a null leaf is inserted to T , we do nothing, so the difficult case is inserting an active leaf.

Suppose that an active leaf $(q, q, 0)$ is inserted to T . Let u be its parent in T . We consider two cases:

1. **u is active.** We query the active tracker data structure: $l_u \leftarrow \text{GETACTIVELEAF}(u, T)$. This means u is an active node, and $(q, q, 0)$ will become the rightmost leaf of u . Thus, we simply insert $(q, q, 0)$ by finding the current rightmost leaf, say x , of u by calling $\text{TRACKRIGHTMOSTLEAF}(u, L_F)$ (in Figure 8) and insert q right after x in σ . By Lemma 11, the running time of this step is $O(\log n)$.
2. **otherwise, u is inactive.** Let $v \leftarrow \text{GETLOWESTACTIVEANCESTOR}(u, T)$. We then examine every child of v to find the *active child* v_x that the DFS visits before u and after other active children of v ; we can afford to do so since v only has $\varepsilon^{-O(\lambda)}$ children. If v_x exists, then we find the rightmost leaf, say x , of v_x by calling $\text{TRACKRIGHTMOSTLEAF}(v_x, \sigma)$ and insert q after x in σ . Note that in the DFS order, x is followed by the active descendant leaves of u in σ . Since $(q, q, 0)$ is the only active leaf in descendants of u , q must be the new successor of x in σ . Otherwise, v_x does not exist. Since v is active, there exists an active child v_y of v such that v_y is visited after u and before other active children of v . We then find the leftmost leaf, say y , of v_y by calling $\text{TRACKLEFTMOSTLEAF}(v_y, L_F)$ and insert q before y in σ .

Since checking if a child of v is active can be done in $O(1)$ time, the total time to find v_x and v_y is $\varepsilon^{-O(\lambda)}$. By Lemma 11, finding the leftmost or the rightmost leaf can be done in $O(\log n)$ time. Thus, the total running time of this step is $O(\varepsilon^{-O(\lambda)} + \log n)$.

By considering all the cases, updating σ can be done in $O(\varepsilon^{-O(\lambda)} + \log n)$.

1562 **5.2.2 Active tracker data structure**

1563 Now, we provide details of active data structures as claimed in Lemma 16. For each centroid path π in
 1564 the centroid decomposition of T , we keep track of the *lowest active node* of π , denoted by $\text{LOWEST}(\pi)$.
 1565 Specifically, the head of π will store a pointer that points to $\text{LOWEST}(\pi)$, so that it takes only $O(1)$ time
 1566 to find $\text{LOWEST}(\pi)$. Initially, $\text{LOWEST}(\pi)$ is `NULL`, then it will be updated while nodes of T are inserted
 1567 or deleted. For now, we assume that $\text{LOWEST}(\pi)$ is given, and we will use it to implement all other
 1568 operations of the active tracker data structure; the pseudocodes are given in Figure 9. We will come
 1569 back to the issue of maintaining $\text{LOWEST}(\pi)$ for every path π later.

FINDCENTROIDPATH(u, T): *find the centroid path containing u*

Let v be the last element of the ancestral array A_u of u , and π_u be the centroid path whose head is v . Then return π_u .

IsActive(u, T):

1. Let $\pi_u \leftarrow \text{FINDCENTROIDPATH}(u, T)$.
2. Let $t \leftarrow \text{LOWEST}(\pi_u)$.
3. If t is `NULL` or the level of t is higher than u , then u is an inactive node, and we return `FALSE`.
 Otherwise, we return `TRUE`.

GetActiveLeaf(u, T):

«« u is guaranteed to be active »»

1. Let $\pi_u \leftarrow \text{FINDCENTROIDPATH}(u, T)$.
2. Let $t \leftarrow \text{LOWEST}(\pi_u)$.
 - (a) if t is a leaf, we return t .
 - (b) Otherwise, we pick an active child t' of t (by maintaining a pointer to an arbitrary active child at t) and return $\text{GetActiveLeaf}(t', T)$.

GetLowestActiveAncestor(u, T): *«« see Figure 10 »»*

1. Let $v \in A_u$ be the lowest head such that its corresponding centroid path π has $\text{LOWEST}(\pi) \neq \text{NULL}$. (We find v by considering every element of A_u .)
2. [Case 1: $u \in \pi$.] If the level of $\text{LOWEST}(\pi)$ is higher than u , we return $\text{LOWEST}(\pi)$. Otherwise, we return u .
3. [Case 2: $u \notin \pi$.] Let j be the index of v in A_u . Let v' be the parent of $A_u[j+1]$. If v' is active, return v' . Otherwise, return $\text{LOWEST}(\pi)$.

Figure 9: Operations supported by the active tracker data structure.

1570 **Correctness.** Since the ancestral array A_u of a node u contains the heads of all centroid paths of u
 1571 sorted by decreasing level, and u belongs to some centroid path, $\text{FINDCENTROIDPATH}(u, T)$ in Figure 9
 1572 correctly returns the centroid path containing u and its head. Clearly, if u is active, then by definition of
 1573 LOWEST , the lowest active node in π_u in line 2 of $\text{IsActive}(u, T)$ has a level at most that of u , and hence

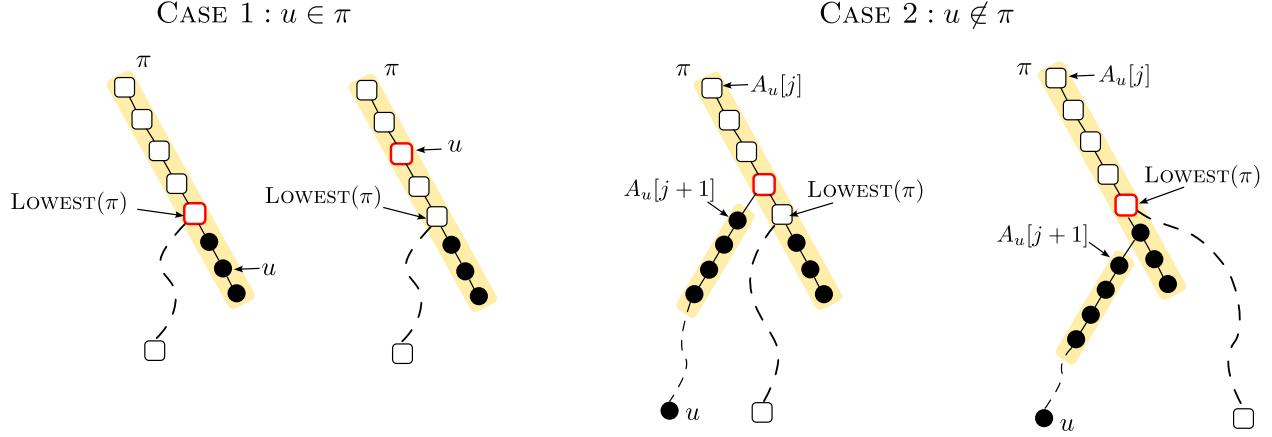


Figure 10: Illustrate GETLOWESTACTIVEANCESTOR operation. Let π be the lowest centroid path of u whose head is active, and j be the index of π 's head in A_u . Rectangular white nodes are active, round black nodes are inactive. The lowest active ancestor of u is the red node. To find the lowest active ancestor, when $u \in \pi$, we consider the levels of u and $LOWEST(\pi)$, when $u \notin \pi$, we consider the levels of $A_u[j+1]$'s parent and $LOWEST(\pi)$.

1574 $\text{ISACTIVE}(u, T)$ correctly decides if u is active or not.

1575 For $\text{GETACTIVELEAF}(u, T)$, the input node u is always an active node. Since u is active, the node t in
 1576 line 2 exists and has a level at most u . If t is a leaf, then it will be an active leaf of u , so the algorithm
 1577 is correct. Otherwise, it recursively finds an active leaf from an active child t' of t . By definition of an
 1578 active node, t' has an active leaf in its subtree, and hence the algorithm is correct.

1579 For $\text{GETLOWESTACTIVEANCESTOR}(u, T)$, case 1 is self-explained. For case 2, by definition of A_u and v ,
 1580 $A_u[j+1]$ is the head of a centroid path, say π' , of u such that every node π' is inactive, and therefore,
 1581 the lowest active ancestor of u must be an ancestor of $A_u[j+1]$. Since v' is the parent of $A_u[j+1]$, if it is
 1582 active, then it is the lowest active ancestor of u , and hence the algorithm is correct. Otherwise, observe
 1583 that v' is in π by the definition of centroid path decomposition. Since v' is in active, every active node
 1584 in π is an ancestor of v' , and hence $LOWEST(\pi)$ is the lowest active ancestor of v' , which is also of u .
 1585 Therefore, the algorithm is correct.

1586 To show Lemma 16, it remains to bound the running time of each operation.

1587 **Proof (Proof of Lemma 16):** Observe that FINDCENTROIDPATH runs in $O(1)$ time and hence ISACTIVE
 1588 also runs in $O(1)$ time.

1589 In GETACTIVELEAF , observe that the algorithm recursively invokes LOWEST on centroid paths of the
 1590 final active leaf, and there are only $O(\log n)$ such paths by definition of the centroid path decomposition.
 1591 As we maintain an active child for every lowest active node of a path, t' in step 2(b) can be found in
 1592 $O(1)$ time, giving $O(\log n)$ total running time to find an active leaf.

1593 In $\text{GETLOWESTACTIVEANCESTOR}$, we can find v in step 1 in $O(\log n)$ time since $|A_u| = O(\log n)$. Other
 1594 steps can be implemented in $O(1)$ time, and hence, the total running time is $O(\log n)$. \square

1595 **Maintaining $LOWEST(\pi)$ pointers.** Cole and Gottlieb [12] described major ideas for maintaining
 1596 $LOWEST(\pi)$ pointers. Here, we review their ideas. We also fill in the missing detail of handling the
 1597 changes to the dynamic centroid path decomposition. Recall that there are three types of simple updates
 1598 to the PIT T : adding a leaf, marking a leaf as deleted, and subdividing an edge. Observe that subdividing
 1599 an edge does not change the active/inactive status of the endpoints of the subdivided edge. Marking a

leaf as deleted could turn some ancestors of the leaf from active to inactive, while adding a leaf could, on the other hand, turn its ancestors from inactive to active.

One subtlety here is that subdividing an edge and adding a leaf could change the size of the tree T and trigger updates to the centroid decomposition of T . These updates are handled by the underlying dynamic data structure for the centroid decomposition. While subdividing an edge does not change the status of all other nodes, adding a new leaf could cause massive changes. Therefore, when adding a new leaf, we will consider it as *inactive* so that the status of all other nodes is unchanged, and hence, the underlying dynamic data structure for the centroid decomposition could proceed as usual. Note that the underlying dynamic data structure for the centroid decomposition now has to maintain an additional pointer $\text{LOWEST}(\pi)$ for centroid path π , and with the assumption that the status of all the nodes is unchanged, this can be done by slightly augmenting the data structure by Kopelowitz and Lewenstein in the previous section. Once the dynamic updates to the structure of the centroid decomposition are done, we change the status of the newly inserted node to active and update the status of other nodes. This is exactly the same problem with marking a leaf as deleted since we simply change the status of this leaf from active to inactive. (We note that the newly inserted leaf could be null, and in this case, there is no need to turn it to active; hence, we only need to consider inserting a non-null leaf.)

Now, we will handle a leaf status change from active to inactive or vice versa. Let π be a centroid path, and t is the current lowest active node of π . Let i be the level of t . If the only active leaf of t changes to inactive, then t becomes inactive. Then we need to find another lowest active node to update $\text{LOWEST}(\pi)$. Observe that the head of π is active if π contains an active leaf or a node with an active child not on π , called an *off-path child*. The idea of Cole and Gottlieb [12] is to keep track of nodes that have an active off-path child, and this task can be efficiently done by using a balanced binary tree. Here, we use a skip list instead of a balanced binary tree for two following reasons: (1) it was not clear how Cole and Gottlieb [12] updated binary trees when their centroid paths change, (2) a list of nodes in π with an active off-path child is a subpath of π , thus we can track these nodes in a skip list by the same way we maintain π , and therefore, we can resolve (1) effectively.

Now consider a centroid path π and a node t of π . If t is an active leaf or t has an active off-path child, then we say t is a *low candidate* of π . Let $B(\pi)$ be the skip list that contains all the low candidates of π , and the keys to $B(\pi)$ are the levels of the candidates. We claim that the lowest node of $B(\pi)$ is $\text{LOWEST}(\pi)$. To see this, let z be the lowest active node of π . If z is a leaf, since z is active, z is a low candidate of π . If z is not a leaf, since the child of z in π is either inactive or null (in the case where z is the low endpoint of π), z must have an active off-path child. Thus, z is also a low candidate of π . In both cases, z is maintained in $B(\pi)$. Therefore, we can update $\text{LOWEST}(\pi)$ by simply taking the lowest node in $B(\pi)$.

Let l be the leaf whose status changes. Since only ancestors of l change status, we only need to update all the centroid paths of l in bottom-up order, and there are only $O(\log n)$ such paths. However, we note that, since a single centroid path could have up to $\Omega(n)$ nodes, a balanced binary tree could incur $O(\log(n))$ time per update, potentially bringing the total update time up to $\Theta(\log^2(n))$. Cole and Gottlieb [12] resolved this issue by observing that if a centroid path π has at least one low candidate, then changing a leaf under π from inactive to active does not change the status of the head v of π . Similarly, if π has more than one low candidate, then changing a leaf under π from active to inactive does not change the status of the head. In both cases, the parent of v is still a low candidate of the *parent path* of π , which is the path containing the parent of π 's head, and therefore, we could terminate the status update at π . For all centroid paths that are descendants of π (which are paths whose heads are descendants of π 's head), we update their skip lists in $O(1)$ since they have at most one low candidate. Thus, in the entire process, π is the one path that could incur in $O(\log(n))$ time to update $B(\pi)$, and therefore the total running time is $O(\log(n))$.

In addition, recall that we maintain a pointer to an arbitrary active off-path child for every low

candidate (see `GETACTIVELEAF`). Let $ACTIVECHILD(t)$ be an active off-path child of a low candidate t ; $ACTIVECHILD(t)$ is `NULL` if t does not have any. We also count the number of active off-path children for each low candidate by $ACTIVECOUNT(t)$. Now we can check in $O(1)$ when t in π is no longer a low candidate to remove t out of $B(\pi)$.

In Figure 11, we show the pseudocode of how to change the status of a leaf from active to inactive, following the discussion above. The code for changing from inactive to active is very similar. At step 2, we add l to $B(\pi)$, assign l to $LOWEST(\pi)$, and terminate if $|B(\pi)| > 1$. We decrease $ACTIVECOUNT(t)$ by 1 at 3(b). We do not find an active off-path child to replace v in step 3(c). We terminate if $ACTIVECOUNT(t) > 1$ at 3(d). At 3(e), we add t to $B(\pi')$ and assign $ACTIVECHILD(t) = v$, $ACTIVECOUNT(t) = 1$. At 3(f), we terminate if $|B(\pi')| \geq 2$.

DEACTIVATE(l, T): change the status of a leaf l from active to inactive

1. Let $\pi \leftarrow \text{FINDCENTROIDPATH}(l, T)$, and v be the head of π .
2. Remove l out of $B(\pi)$ and update $LOWEST(\pi)$ by getting the lowest node in $B(\pi)$. If $|B(\pi)| > 0$, v is still active after deleting l , we terminate.
3. Repeat the following steps as long as the head of π is not the root of T :
 - (a) Let π' be the parent centroid path of π , and t be the parent of v in π' .
 - (b) Decrease $ACTIVECOUNT(t)$ by 1.
 - (c) If $ACTIVECHILD(t) = v$, we find another active off-path child to replace v , or assign `NULL` to this pointer if t does not have any.
 - (d) If $ACTIVECOUNT(t) > 0$, t is still a low candidate of π , we terminate.
 - (e) Otherwise, t is no longer a low candidate of π' after deleting l .
 - [Update $B(\pi')$.] Remove t out of $B(\pi')$, the skip list maintaining low candidates of π' .
 - [Update $LOWEST(\pi')$.] Find $LOWEST(\pi')$ by getting the lowest node in $B(\pi')$.
 - [Update t .] $ACTIVECHILD(t) \leftarrow \text{NULL}$ and $ACTIVECOUNT(t) \leftarrow 0$
 - (f) If $|B(\pi')| \geq 1$, terminate. In this case, the head of π' remains active after deleting l , and deleting l does not change the low candidates of ancestor paths of π' .
 - (g) Update $\pi \leftarrow \pi'$.

Figure 11: Changing status of a leaf l from active to inactive.

Observation 4. ACTIVATE runs in $O(\log(n))$ time, DEACTIVATE runs in $O(\varepsilon^{-O(\lambda)} + \log(n))$ time.

Proof: Updating t to $B(\pi')$ in step 3 costs the most running time, which is $O(\log(n))$ time if $B(\pi')$ has at least one low candidate, and $O(1)$ time if it has one candidate. There are at most $O(\log(n))$ centroid paths, and once the algorithm considers a path π' that has more than one low candidate, it terminates. Thus, the cost of adding a node to a skip list is $O(1)$ for each descendant centroid path of π' , and $O(\log(n))$ only for π' . Therefore, ACTIVATE runs in $O(\log(n))$ time.

For DEACTIVATE , it may have to update $ACTIVECHILD$ at step 3(c). Observe that there is at most one centroid path π' such that $ACTIVECOUNT(t) > 0$ occurs, thus the running time of step 3(c) is $O(\varepsilon^{-O(\lambda)})$ when the algorithm considers π' , and $O(1)$ when the algorithm considers descendant centroid paths of π' . Therefore, DEACTIVATE runs in $O(\varepsilon^{-O(\lambda)} + \log(n))$ time. \square

1668 6 Dynamic Net Tree

1669 In this section, we show how to maintain a net tree for a dynamic point set S as described in Theorem 10,
 1670 which we restate below:

1671 **Theorem 10.** *Given $b \geq 2$ a parameter of the jump isolation, $\varepsilon \leq \frac{1}{4b}$, there is a data structure maintaining
 1672 a (δ, ε) -net tree T such that T has the following properties:*

- 1673 • [Packing.] Two nodes at the same level (x, i) and (y, i) have $d_X(x, y) > \frac{1}{4} \frac{\delta}{\varepsilon^i}$.
- 1674 • [Covering.] If (x, i) is the parent of (y, i') where $i' < i$, then $d_X(x, y) \leq \phi \frac{\delta}{\varepsilon^i}$, where $\phi = \frac{3}{4}$.
- 1675 • [b-Jump isolation.] Any jump is **b-isolated**: given a jump starting from a node (x, i) , for any node
 1676 (y, k) who is not a descendant of (x, i) for $k < i$, $d_X(x, y) > b \frac{\delta}{\varepsilon^k}$.
- 1677 • [Close-containment.] For any (y, k) and any ancestor (z, i) of (y, k) , $d_X(y, z) \leq \frac{\delta}{\varepsilon^i} - \frac{\delta}{\varepsilon^k}$. This
 1678 implies that every point p in the subtree rooted at (z, i) is contained in $B(z, \frac{\delta}{\varepsilon^i})$, i.e., $d_X(p, z) \leq \frac{\delta}{\varepsilon^i}$.

1679 Furthermore, given access to a node (x, i) in T at level i , if (x, i) is not the bottom node or a hidden node
 1680 in a jump, then we can find all the nodes (y, i) at level i such that $d_X(x, y) = g \cdot \frac{\delta}{\varepsilon^i}$ for any parameter
 1681 $g \geq 1$ in $O(g)^\lambda$ time. The data structure has space $O(n)$ and runs in $O_\lambda(\log n)$ time per update.

1682 Recall that a jump from (x, i) down to (x, j) for $i > j + 1$ intuitively hides nodes $(x, j + c)$ for
 1683 $j < j + c < i$, and we call the nodes $(x, j + c)$ **hidden nodes**. In a net tree cover, sometimes we merge two
 1684 nodes at level i if their distance is at most $3c \cdot \frac{\delta}{\varepsilon^i}$ (they are $3c$ -close) for some parameter c . Therefore,
 1685 we need to guarantee that a hidden node is not $3c$ -close to any existing node at the same level, which
 1686 inspires the jump isolation property.

1687 Previously, Cole and Gottlieb [12] constructed a net tree for a dynamic point set with $O_\lambda(\log n)$ time
 1688 per update and $O(n)$ space. Specifically, they maintained the covering and jump isolation properties
 1689 for the net tree by modifying only $O_\lambda(1)$ nodes per update. To this end, they introduced the concept
 1690 of *rings* and used 5 rings in their construction. Here, we simplify their construction and the analysis.
 1691 We observe that Cole and Gottlieb [12] used one ring among the five for the search operation and four
 1692 rings to consider the distance of a node to its descendants. We simplify their insert operation, which
 1693 is the bulk of the technical details, and our simplified operation only requires four rings. Furthermore,
 1694 we associate each ring in our construction with specific functionality, making our overall construction
 1695 simpler and more intuitive.

1696 We remark that our close-containment property in Theorem 10 is more relaxed (which makes it
 1697 easier to guarantee) than that of Cole and Gottlieb [12]. Specifically, in their work, for any node (y, k)
 1698 and any ancestor (z, i) of (y, k) has $d_X(z, y) < \frac{4}{5} \frac{\delta}{\varepsilon^i} - \frac{\delta}{\varepsilon^k}$, which is smaller than the upper bound in our
 1699 close-containment property. Our relaxation is due to the difference in the way we set up the parameters
 1700 of packing and covering properties.

1701 **Overview of the dynamic net tree.** We now sketch the structure of the dynamic net tree with the
 1702 covering, jump isolation, and close-containment properties. This structure was largely developed by
 1703 Cole and Gottlieb [12]; we simplify some parts that we will detail along the way. It is useful to think of
 1704 each node (x, i) in the net tree as associated with a ball $B(x, \frac{\delta}{\varepsilon^i})$; all points under the subtree rooted at
 1705 (x, i) must be contained in this ball by the close-containment property. (Sometimes, we use the node and
 1706 ball terminologies in the net tree interchangeably.) In addition, the close-containment property implies
 1707 that if (x, i) is an ancestor of (y, j) for $j < i$, then $B(y, \frac{\delta}{\varepsilon^j}) \subseteq B(x, \frac{\delta}{\varepsilon^i})$. We note that it is not so hard to
 1708 show that the covering property implies the close-containment property when ε is sufficiently small (see
 1709 Lemma 19).

1710 For the jump isolation property, recall that a jump from (x, i) down to (x, j) for $i > j + 1$ is a long
 1711 edge connecting two nodes of the same point x at level i and j . Look at two corresponding balls centered
 1712 at x : $B(x, \frac{\delta}{\varepsilon^i})$ and $B(x, \frac{\delta}{\varepsilon^j})$. The jump isolation means that any point at level $k < i$ outside the smaller
 1713 ball, which is $B(x, \frac{\delta}{\varepsilon^j})$, must be at a distance more than $b \frac{\delta}{\varepsilon^k}$ from x . Note that the dynamic net tree of
 1714 Cole and Gottlieb [12] also has a similar structure but with a slightly different covering property and the
 1715 jump isolation property.

1716 Whenever a new point q is inserted, we first need to find the parent node (t, i) for q at some level i .
 1717 Once (t, i) is found, we create a new node $(q, i - 1)$ as a child of (t, i) , and finally create a jump from
 1718 $(q, i - 1)$ down to $(q, 0)$. Sometimes, we have to break a jump, which means adding (t, i) in the middle
 1719 of the edge between (t, j) and (t, k) for $j < i < k$. For a node (t, i) to be the parent of q , it has to satisfy
 1720 the covering property. We could find (t, i) by visiting the dynamic net tree level-by-level, but doing so
 1721 would result in a running time to the height of the tree, which can be up to $\Omega(n)$. Instead, we find (t, i)
 1722 in two steps: (i) we first find a node (t', i') which is closer to the true parent (t, i) of q , and (ii) then
 1723 we find (t, i) “around the neighborhood” of (t', i') . The node (t', i') we found in the first step has i' to
 1724 be the lowest level such that $d_X(q, t') \leq \frac{\delta}{\varepsilon^{i'}}$. This means (t', i') and $(q, 0)$ satisfy the close-containment
 1725 property, which is a relaxation of the covering property. Therefore, we could apply a binary search to
 1726 find (t', i') via the so-called *containment search* introduced by Cole and Gottlieb [12]. Here, we slightly
 1727 modify the search condition in the containment search to fit our purpose. For step (ii), we show that
 1728 (t', i') is very close to the parent node (t, i) that we are searching for, and hence we only need to spend
 1729 $O_\lambda(1)$ additional time to locate (t, i) . We also simplify several steps to find (t, i) from (t', i') .

1730 A very important subtlety in the search for the parent node (t, i) of a newly inserted point q is that
 1731 when (t, i) is found, $d_X(t, q)$ might be more than $\phi \cdot \delta / \varepsilon^i$ so that the covering property would be violated
 1732 at level i if we made $(q, i - 1)$ a child of (t, i) . The idea to resolve this issue is to *promote* (t, i) to the
 1733 next level (the pseudocode in Figure 13), to become $(t, i + 1)$ so that at level $i + 1$, $d_X(t, q) \leq \phi \cdot \delta / \varepsilon^{i+1}$
 1734 and hence we can make a child node (q, i) of $(t, i + 1)$ without violating the packing property. However,
 1735 promoting (t, i) to $(t, i + 1)$ might lead to another violation of the covering property between $(t, i + 1)$
 1736 and its parent at level $i + 2$, which requires another promotion to resolve and consequently triggers a
 1737 chain many promotions. Cole and Gottlieb [12] resolved this issue with *rings* and used five rings in their
 1738 construction. Here, we only use four rings, from innermost to outermost, and consider which ring the
 1739 distance between a node and its parent falls into to determine whether a node should be promoted. If
 1740 the distance is in ring 4 (the outermost ring), then we will promote the node so that it belongs to the first
 1741 three (inner) rings, and hence no further promotion is needed for maintaining the covering property.

1742 We now describe the details of the dynamic net tree, which are organized as follows. In Section 6.1,
 1743 we describe the idea of rings. Then in Section 6.2, we show the containment search and operations
 1744 designed by Cole and Gottlieb [12] that support the dynamic net tree construction. The full construction
 1745 with our modifications for search and insertion is shown in Section 6.3. Finally, in Section 6.4, we
 1746 prove Theorem 10.

1747 6.1 Rings

1748 To define rings, we will use the following constants:

$$1749 \alpha = \frac{1}{4}, \beta = \frac{2}{4}, \phi = \frac{3}{4}, \gamma = 1, \psi = \frac{5}{4}, \text{ and } \varepsilon \leq \frac{\alpha}{b} \quad (6)$$

1750 where α is the constant in packing distance, ϕ is the constant in the covering distance, γ is the constant
 1751 in containment distance, b is the parameter for jump isolation, and ψ is a constant in a search operation
 1752 (we will see later in Section 6.3). Given $b \geq 2$, we have $\varepsilon \leq \frac{1}{8}$.

1753 The values of constants in Equation (6) satisfy the following inequalities:

1754 1. $\psi\varepsilon + \beta \leq \phi$, $\varepsilon\psi + \alpha \leq \beta$ and $\psi\varepsilon \leq \beta$. These conditions are to maintain the covering property.

1755 2. $\phi \leq \gamma(1 - \varepsilon)$. This condition is to maintain the close-containment property.

1756 3. $\psi\varepsilon \leq \alpha$, $\alpha + \gamma \leq \psi \leq b$, $\varepsilon \leq \frac{\alpha}{b}$. These conditions are to maintain the jump isolation.

1757 Note that Cole and Gottlieb [12] did not parameterize the jump isolation property; their construction
1758 only gives $b = 2$. Their values of the five other constants respectively were $\alpha = \frac{1}{5}$, $\beta = \frac{2}{5}$, $\phi = \frac{3}{5}$, $\gamma = \frac{4}{5}$,
1759 $\psi = 1$, and $\varepsilon \leq \frac{1}{5}$. The inequalities above also hold with their values. They used 5 rings defined by 5
1760 constants above, while we have 4 rings with 4 parameters $\alpha, \beta, \phi, \gamma$.

1761 Our rings are formally defined as follows:

- 1762 • Ring⁹ α of a ball $B(x, \frac{\delta}{\varepsilon^i})$ is $B(x, \alpha \frac{\delta}{\varepsilon^i})$. If $p \in B(x, \alpha \frac{\delta}{\varepsilon^i})$,
1763 we say p is *in ring α* of (x, i) .
- 1764 • Ring β of a ball $B(x, \frac{\delta}{\varepsilon^i})$ is $B(x, \beta \frac{\delta}{\varepsilon^i}) \setminus B(x, \alpha \frac{\delta}{\varepsilon^i})$. If
1765 $p \in B(x, \beta \frac{\delta}{\varepsilon^i}) \setminus B(x, \alpha \frac{\delta}{\varepsilon^i})$, we say p is *in ring β* of (x, i) .
- 1766 • Ring ϕ of a ball $B(x, \frac{\delta}{\varepsilon^i})$ is $B(x, \phi \frac{\delta}{\varepsilon^i}) \setminus B(x, \beta \frac{\delta}{\varepsilon^i})$. If
1767 $p \in B(x, \phi \frac{\delta}{\varepsilon^i}) \setminus B(x, \beta \frac{\delta}{\varepsilon^i})$, we say p is *in ring ϕ* of
1768 (x, i) .
- 1769 • Ring γ of a ball $B(x, \frac{\delta}{\varepsilon^i})$ is $B(x, \gamma \frac{\delta}{\varepsilon^i}) \setminus B(x, \phi \frac{\delta}{\varepsilon^i})$. If
1770 $p \in B(x, \gamma \frac{\delta}{\varepsilon^i}) \setminus B(x, \phi \frac{\delta}{\varepsilon^i})$, we say p is *in ring γ* of (x, i) .

1771 Given s in the set $\{\alpha, \beta, \phi, \gamma\}$, we say a point p is *completely out of ring s* of $B(x, \frac{\delta}{\varepsilon^i})$ if $p \notin B(x, s \frac{\delta}{\varepsilon^i})$. Note that the
1772 notion of completely out of ring s only applies to a point
1773 p where $d_X(p, x) > s \frac{\delta}{\varepsilon^i}$, not to a point q where it is inside
1774 $B(x, s \frac{\delta}{\varepsilon^i})$ and does not belong to ring s . As the rings can be
1775 linearly ordered by increasing radii, if a point is completely
1776 out of ring ϕ , say, then it is also completely out of rings α
1777 and β .

1778 We classify nodes based on their distances to parents. Given a node (t, j) , we say that (t, j) is a *ring- s node*
1779 if t is in ring s of its parent. The following observations follow from the definition of a jump and
1780 the covering property.

1781 **Observation 5.** (a) A node is either a *ring- α* , *ring- β* , or *ring- ϕ* node.

1782 (b) A node at the bottom of a jump or created in the middle of a jump is a *ring- α* node.

1783 (c) A *ring- β* or *ring- ϕ* node at level i has a parent at level $i + 1$.

1785 6.2 Containment Search and Internal Operations

1786 In this section, for completeness, we briefly describe the dynamic net tree of Cole and Gottlieb [12].
1787 We also provide the details of operations and data structures that we reuse or modify to construct our
1788 dynamic net tree. The main goal of Cole and Gottlieb in [12] is to develop a dynamic data structure for
1789 solving the approximate nearest neighbor search problem with $O_\lambda(\log(n))$ time per update and query. To
1790 this end, they developed two data structures: (i) a graph (might not be a tree) to maintain (a hierarchy
1791 of) nets with the packing, covering, and jump isolation properties and, (ii) a spanning of the graph and a

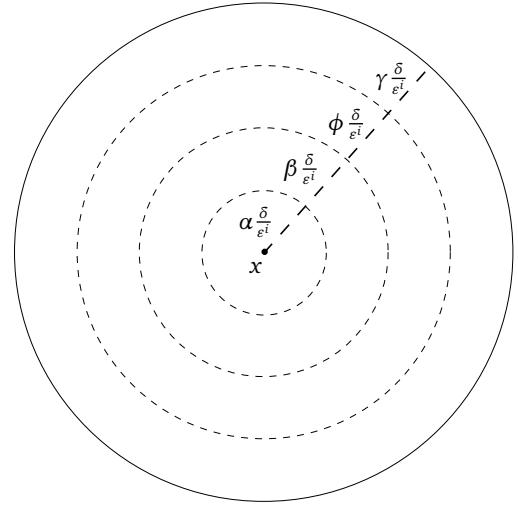


Figure 12: Rings of $B(x, \frac{\delta}{\varepsilon^i})$

⁹More precisely, ring α is a ball, but we use ring terminology to be consistent with other rings.

1792 central path decomposition on top of the spanning to quickly implement a procedure called *containment*
 1793 *search* (which we will give more details below).

1794 One can extract from their data structures a dynamic net tree supporting containment search as
 1795 stated below. Each node at level i of the net tree is associated with a ball of radius δ/ϵ^i centered at the
 1796 point in the node.

1797 **Definition 13 (Containment search data structure of Cole and Gottlieb [12]).** *There is a data struc-
 1798 ture that maintains a net tree T and supports the following operations:*

- 1799 • **CONTAINMENTSEARCH(q, T):** *If q is an existing point, return the leaf $(q, 0)$. If q is a new point,
 1800 return the lowest ball containing q . That is, it returns the node (t, i) such that $d_X(t, q) \leq \frac{\delta}{\epsilon^i}$ and
 1801 for all $j < i$, there is no (z, j) with $d_X(z, q) \leq \frac{\delta}{\epsilon^j}$. This operation runs in $O_\lambda(\log n)$ time.*
- 1802 • **INSERT(q, T):** *create (several) nodes associated with q in T . This operation invokes $\text{SEARCH}(q, T)$
 1803 and executes $O_\lambda(1)$ additional basic operations; thus it runs in $O_\lambda(\log n)$.*
- 1804 • **DELETE(q, T):** *mark $(q, 0)$ as deleted in $O_\lambda(\log n)$ time.*

1805 One key technical idea in the work of Cole and Gottlieb [12] was to search the parent for a new
 1806 point in roughly $O(\log n)$ time using containment search, as described in details below. When a new
 1807 point q is added to S , the insert operation will invoke the search containment to find a node that is
 1808 close to the parent node. From there, in $O(1)$ additional steps, they can find the exact parent (and then
 1809 modify the tree). When a point is removed from S , they mark the corresponding leaf as deleted. After a
 1810 predefined number of deletions, they rebuilt the data structure in the background to remove the nodes
 1811 associated with deleted points completely (and also to de-amortize). A subtle issue is that the dynamic
 1812 net tree might contain deleted points since deletions are only marked, while the containment search has
 1813 to return non-deleted points. Cole and Gottlieb[12] resolved this issue by spending an extra $O(\log(n))$
 1814 time per deletion.

1815 In addition to containment search, the insert operation invokes several other operations, called
 1816 *internal operations*, to modify the net tree. These include creating a new node, promoting a node to
 1817 maintain the covering property, creating or splitting a jump, and fixing a jump to maintain the jump
 1818 isolation property. We remark that the jump isolation property is important to the correctness of the
 1819 containment search operation in Definition 13.

1820 Our simplified insert operation given in the next section reuses most of the internal operations stated
 1821 in this section and only modifies the PROMOTE operation. Therefore, we can still apply the containment
 1822 search of Cole and Gottlieb [12] to construct our dynamic net tree.

1823 Now, we describe the containment search and the internal operations by Cole and Gottlieb [12]

1824 **Containment search.** Recall that to maintain the covering property, whenever a point q is added,
 1825 we need to find a node (t, i) such that (t, i) and $(q, i-1)$ satisfy $d_X(q, t) \leq \phi \frac{\delta}{\epsilon^i}$ so that we could make
 1826 $(q, i-1)$ as a child of (t, i) without violating the covering property. Finding (t, i) directly is difficult since
 1827 the distance upper bound $\phi \frac{\delta}{\epsilon^i}$ is very tight. Instead, Cole and Gottlieb [12] relaxed this upper bound
 1828 to $\frac{\delta}{\epsilon^i}$, which is exactly the close-containment property. The relaxed upper allows them to apply binary
 1829 search on (the centroid-path decomposition of) the net tree. This is because by the close-containment
 1830 property, any ancestor (z, k) of (t, i) has $d_X(z, q) \leq \frac{\delta}{\epsilon^k}$ and hence every node on the path from (t, i) to
 1831 the root satisfies the close-containment property with respect to $(q, 0)$, which is ideal for binary search.
 1832 The binary search returns either (t, i) , which is the node that they are looking for, or a node at a level
 1833 lower than i . In the latter case, they will spend $O(1)$ extra steps to find (t, i) . In both cases, they could
 1834 find the parent for q in total $O_\lambda(\log(n))$ time.

1835 Next, we describe the details of the binary search inside containment search. Recall that the goal is,
 1836 given a point q , to return the lowest ball containing q , i.e., the node (t, i) such that $d_X(t, q) \leq \frac{\delta}{\varepsilon^i}$ and
 1837 for all $j < i$, there is no (z, j) with $d_X(z, q) \leq \frac{\delta}{\varepsilon^j}$, in $O_\lambda(\log n)$ time. Cole and Gottlieb [12] maintained
 1838 a centroid-path decomposition of the net tree, which partitions the tree into a set of paths; each path
 1839 is stored as a skip list. Then, binary search is applied to each centroid path, starting from the path π
 1840 containing the root of the tree. It returns the lowest node (s, j) on π such that $B(s, \frac{\delta}{\varepsilon^j})$ contains q . The
 1841 algorithm then examines (s, j) and its nearby nodes: either some of them is the lowest ball containing
 1842 q , or there is a child $(r, j-1)$ of a node among them such that $B(r, \frac{\delta}{\varepsilon^{j-1}})$ contains q . In the former
 1843 case, we are done, and the containment search terminates. In the latter case, they switched to the path
 1844 (in the centroid-path decomposition) containing $(r, j-1)$ to continue the search. The centroid-path
 1845 decomposition guarantees that switching to a new path reduces half of the nodes in consideration, and
 1846 therefore, the running time of the containment search is $O_\lambda(\log(n))$ in total.

1847 **Remark 5.** Recall that containment search for a point q returns the lowest node (t, i) such that $d_X(t, q) \leq$
 1848 $\frac{\delta}{\varepsilon^i}$. The same idea could be applied to find the lowest (t, i) such that $d_X(t, q) \leq c \cdot \frac{\delta}{\varepsilon^i}$ as long as $c \geq 1$.
 1849 Indeed, we will apply this variant of containment search in our dynamic insertion.

ADD(u, q, T): add q as a child of u

Given a node $u = (t, i)$, we create $(q, i-1)$ as a child of (t, i) then return $(q, i-1)$.

PROMOTE(t, i, T): promote (t, i) , possibly add $(t, i+1)$

«PROMOTE is invoked only when (t, i) is a ring- ϕ node.»

Given a node (t, i) , let $(u, i+1)$ be the current parent of (t, i) ; $(u, i+1)$ exists by Observation 5(c).

1. **[Check the packing property.]** We check the points in Y_{i+1} that are at a distance within $2 \cdot \frac{\delta}{\varepsilon^{i+1}}$ to u and find u' closest to t . If $d(t, u') \leq \alpha \frac{\delta}{\varepsilon^{i+1}}$, we change the parent of (t, i) from $(u, i+1)$ to $(u', i+1)$ and terminate.
2. **[Promote.]**: If $d(t, u') > \alpha \frac{\delta}{\varepsilon^{i+1}}$ (the packing property holds), we create $(t, i+1)$ as a new parent of (t, i) . Next, we find a node at level $(i+2)$ to be the parent of $(t, i+1)$.
3. **[Find parent for $(t, i+1)$.]**: let $(v, i+2)$ be the parent of $(u, i+1)$, then consider nodes within $2 \frac{\delta}{\varepsilon^{i+2}}$ to find $(v', i+2)$ closest to t . By Observation 6, $(v', i+2)$ is closest to t among nodes at level $i+2$.
 - (a) If t is in ring α or ring β of $(v', i+2)$, we choose $(v', i+2)$ to be the parent of $(t, i+1)$.
 - (b) Otherwise, we choose $(v, i+2)$ to be the parent of $(t, i+1)$.

A corner case is when $(u, i+1)$ is the bottom of a jump starting at (u, l) , then $(v, i+2) = (u, i+2)$ is a hidden node. In this case, we create $(u, i+2)$ by invoking JUMPSPLIT($u, l, i+1, i+2, T$), and then proceed as above.

Figure 13: ADD and PROMOTE.

1850 **Internal operations.** These are operations that modify the current net tree to create a new node or
 1851 to maintain the covering and jump isolation properties. In Figure 13, we describe ADD and PROMOTE.
 1852 ADD(u, q, T) simply adds a new node $(q, i-1)$ as a child of a node $u = (t, i)$ at level i . PROMOTE(t, i, T)
 1853 promotes a node (t, i) at level i to $(t, i+1)$ at the next level; see Figure 14. Note that we only invoke
 1854 PROMOTE(t, i, T) when (t, i) is a ring- ϕ node.

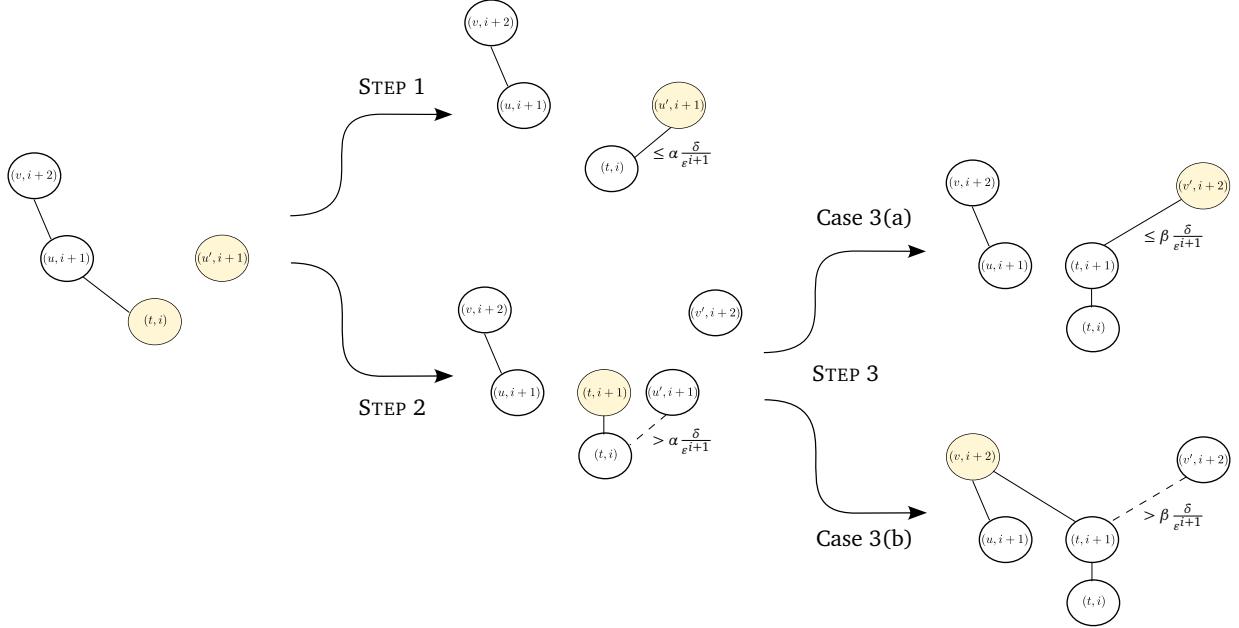


Figure 14: Illustrating PROMOTE operation.

1855 In Step 3 of PROMOTE, given $(v, i + 2)$, we determine $(v', i + 2)$ closest to t among *nearby nodes* of
 1856 $(v, i + 2)$, where $(y, i + 2)$ is a nearby node of $(v, i + 2)$ if $d_X(v, y) \leq 2 \cdot \frac{\delta}{\varepsilon^{i+2}}$. This can be done in $O_\lambda(1)$ time
 1857 by maintaining pointers of nearby nodes for every node in the tree as done in Cole and Gottlieb [12]; by
 1858 the packing bound, there are only $O_\lambda(1)$ nearby nodes. (Every time we add a new node, we will also
 1859 add pointers to nearby nodes following Cole and Gottlieb [12], and hence, in the pseudocodes below,
 1860 we do not include this detail for a cleaner presentation.) We observe that:

1861 **Observation 6.** $(v', i + 2)$ is closest to t among all nodes at level $i + 2$.

1862 **Proof:** Before $\text{PROMOTE}(t, i, T)$, $(v, i + 2)$ is the ancestor at level $i + 2$ of (t, i) . By the close-containment
 1863 property, $d_X(t, v) \leq \frac{\delta}{\varepsilon^{i+2}}$. If $(v', i + 2)$ is closest to t among nodes at level $i + 2$, then $d_X(t, v') \leq d_X(t, v) \leq$
 1864 $\frac{\delta}{\varepsilon^{i+2}}$, which implies $d_X(v, v') \leq 2 \frac{\delta}{\varepsilon^{i+2}}$ by the triangle inequality. \square

1865 The idea of using $\text{PROMOTE}(t, i, T)$ is to turn a ring- ϕ node to a ring- α node as in the following
 1866 observation.

1867 **Observation 7.** If (t, i) is a ring- ϕ node, then after $\text{PROMOTE}(t, i, T)$, (t, i) becomes a ring- α node.

1868 **Proof:** In step 1, we check if (t, i) is in ring- α of u' and in this case, it becomes a child of u' and hence a
 1869 ring- α node. In step 2, (t, i) becomes a child of $(t, i + 1)$ and hence is also a ring- α node. \square

1870 We remark that in the work of Cole and Gottlieb [12], the definition of PROMOTE was not precisely
 1871 given; they just considered whether we could promote (t, i) without violating the packing property and
 1872 assigned the closest node at level $i + 2$ to be the parent of $(t, i + 1)$ if (t, i) was promoted. Here, we
 1873 consider more cases to find an appropriate parent for $(t, i + 1)$; this is necessary to simplify the insertion
 1874 operation.

1875 Figure 15 describes the operations to create a jump or maintain jump isolation. The operation
 1876 $\text{MAINTAINJUMPISOLATION}$ checks if a jump is isolated based on the definition of jump isolation below.

JUMPCREATE(u, T): *create a leaf and a jump u to the leaf*

Given a node $u = (q, i)$, we create a node $(q, 0)$ as a child of (q, i) .

JUMPSPLIT(t, i, j, k, T): *split the jump from (t, i) to (t, j) by inserting (t, k) and possibly $(t, k-1)$*

Given a jump starting from (t, i) down to (t, j) , a level k where $j < k < i$, we create a node (t, k) as a child of (t, i) , create $(t, k-1)$ as a child of (t, k) if $(t, k-1)$ does not exist, and change the parent of (t, j) to $(t, k-1)$.

JUMPFIX(t, i, T): *fixing jump isolation property at (t, i)*

Given a jump starting from (t, i) down to (t, j) , if the jump isolation at (t, i) is violated, we create $(t, i-1)$ as the only child of (t, i) and make it the parent of (t, j) .

MAINTAINJUMPIISOLATION(t, i, T): *check and fix jump isolation property of jumps nearby (t, i)*

Given a node (t, i) , we find (v, i) where $d_X(v, t) \leq (\alpha + \gamma) \frac{\delta}{\epsilon^i}$. If (v, i) exists, then:

1. If (v, i) is the top of a jump, which means the jump isolation at (v, i) is violated, we invoke **JUMPFIX**(v, i, T) to create $(v, i-1)$.
2. If (t, i) is the top of a jump, which means the jump isolation at (t, i) is violated, we invoke **JUMPFIX**(t, i, T) to create $(t, i-1)$.

Figure 15: Operations to create jumps and maintain jump isolation.

Definition 14 (Jump isolation [12], invariant 1). A jump from (x, i) down to (x, j) is *isolated* if for any (y, k) where $k \leq i$ and (y, k) is not a descendant of (x, i) , $d_X(x, y) > \alpha \frac{\delta}{\epsilon^i} + \gamma \frac{\delta}{\epsilon^k}$.

If jump isolation is violated, it will call **JUMPFIX** to fix the jump isolation property. Basically, **JUMPFIX**(t, i, T) will create a new child $(t, i-1)$ of the node (t, i) , which is the top a jump from (t, i) to (t, j) . By adding $(t, i-1)$, (t, i) is no longer the top of a jump—instead, $(t, i-1)$ now becomes the top—and hence the jump isolation property does not apply (t, i) by definition. We will show in Lemma 21 that jump isolation in Definition 14 implies our b -jump isolation property in Theorem 10 when ϵ is sufficiently small, and therefore, we could reuse all operations in Figure 15 to maintain our b -jump isolation property.

Operations ADD, PROMOTE, JUMPSPLIT, JUMPFIX run in $O_\lambda(1)$ time; each of them will add nodes associated with a new point and modify the current net tree. Accordingly, we will have different types of nodes created or modified by one insertion. These are the types of nodes mentioned in Definition 7 given in Section 4.1.

Find nearby nodes. Given a node (x, i) that is not the bottom or hidden node of a jump, we want to find nodes (y, i) such that $d_X(x, y) \leq g \frac{\delta}{\epsilon^i}$ for a constant g . First, we apply the same idea of Cole and Gottlieb [12]: maintain pointers for every (q, i) to nodes (p, i) if $d_X(p, q) \leq \frac{2\delta}{\epsilon^i}$, and (q, i) is not the bottom or hidden node of a jump. After that, we follow pointers of (x, i) to visit nodes (y_0, i) within distance $2 \frac{\delta}{\epsilon^i}$ from (x, i) , then follow pointers of (y_0, i) to visit nodes (y_1, i) within distance $4 \frac{\delta}{\epsilon^i}$ from (x, i) , and so on. After $O(g)$ steps, we reach nodes (y, i) whose $d_X(x, y) \leq g \frac{\delta}{\epsilon^i}$.

To maintain pointers for every node (q, i) to any (p, i) where $d_X(q, p) \leq \frac{2\delta}{\epsilon^i}$, we find (p, i) when (q, i) is added to the tree. Specifically, after **INSERT**, (q, i) is added to the tree as a child of (t, i') for $i' > i$. If $i' > i + 1$, we do nothing. If $i' = i + 1$, we follow pointers of $(t, i + 1)$ to find $(v', i + 1)$ whose

1899 $d_X(t, v') \leq 2\frac{\delta}{\varepsilon^{i+1}}$. Then for every child (p', i) of $(v', i+1)$, if $d_X(q, p') \leq \frac{2\delta}{\varepsilon^i}$, we create pointers from
1900 (p', i) to (q, i) and (q, i) to (p', i) .

1901 To see the correctness of this idea, we consider the parents of (q, i) and (p, i) . If $i' > i+1$, then (t, i') is
1902 the top of a jump, and (q, i) is the bottom node or in the middle of that jump (in this case, (q, i) is created
1903 by splitting a jump). By the b -jump isolation property, any node (y, i) has $d_X(q, y) > b\frac{\delta}{\varepsilon^i} \geq 2\frac{\delta}{\varepsilon^i}$, thus (q, i)
1904 has no pointers to nearby nodes. When $i' = i+1$, observe that (p, i) is also not the bottom node or in the
1905 middle of a jump. Consider the parent $(v, i+1)$ of (p, i) , we have $d_X(t, v) \leq d_X(q, p) + d_X(q, t) + d_X(v, p)$.
1906 By the covering property, $d_X(q, t)$ and $d_X(v, p)$ are at most $\phi\frac{\delta}{\varepsilon^{i+1}}$, thus $d_X(t, v) \leq 2\frac{\delta}{\varepsilon^i} + 2\phi\frac{\delta}{\varepsilon^{i+1}} \leq 2\frac{\delta}{\varepsilon^{i+1}}$
1907 since $\phi = \frac{3}{4}$ and $\varepsilon \leq \frac{1}{8}$. Therefore, from the parent $(t, i+1)$ of (q, i) , it suffices to consider children of
1908 $(v, i+1)$ whose $d_X(v, t) \leq 2\frac{\delta}{\varepsilon^i}$ to find (p, i) nearby (q, i) .

1909 Back to finding nodes (y, i) whose $d_X(x, y) \leq g\frac{\delta}{\varepsilon^i}$, the process runs in $O(g)$ steps. By the packing
1910 property, each node has $O_\lambda(1)$ pointers. Therefore:

1911 **Claim 2.** *Given (x, i) that is not the bottom or a hidden node of a jump, we can find (y, i) where
1912 $d_X(x, y) \leq g\frac{\delta}{\varepsilon^i}$ from (x, i) with $O(g)^\lambda$ time.*

1913 6.3 Dynamic Net Tree Operations

1914 We are now ready to construct our dynamic net tree. We use the same containment search of Cole and
1915 Gottlieb [12] to search for a slightly different variant of the lowest ball containing q :

CONTAINMENTSEARCH(q, T)

Given a new point q , we apply the containment search of Cole and Gotlieb [12], to find the
lowest node (t, i) in T such that:

$$1916 \quad d_X(q, t) \leq \psi \frac{\delta}{\varepsilon^i} \quad (7)$$

where $\psi = \frac{5}{4}$ (in Equation (6)); see Remark 5. If more than one node at the same level satisfies Equation (7), we return the node closest to p .

1916 Note that the containment search might return a node (t, i) such that (t, i) and $(q, i-1)$ do not
1917 satisfy the covering property. We will handle this case later in the insert operation described below.

1918 **Insertion.** Given a new point q , our goal is to find a parent for q . First, we will invoke CONTAIN-
1919 MENTSEARCH(q, T) to find the lowest node (t, i) satisfying Equation (7). If $d_X(q, t) \leq \phi\frac{\delta}{\varepsilon^i}$, we can make
1920 $(q, i-1)$ a child of (t, i) . In the complementary case, making $(q, i-1)$ a child of (t, i) will violate the
1921 covering property, and hence we have to find another node to be the parent of q . To quickly find the
1922 parent for q , we use the idea of *chains* and *obligations* introduced by Cole and Gottlieb [12]. Chains
1923 and obligations also have another very important role, specifically in avoiding a cascading sequence of
1924 promotions described in the overview.

1925 **Definition 15 (Chain).** *Given a node (t, i) , let (u, j) be the lowest ring- α node that is an ancestor of
1926 (t, i) ; it is possible that $(u, j) = (t, i)$. We define the *chain* of (t, i) to be the sequence of nodes that starts
1927 at (u, j) and ends at (t, i) .*

1928 Observe by Definition 15, if (t, i) is a ring- α node, then the chain of (t, i) has only one node, which
1929 is (t, i) . A chain of a ring- β or ring- ϕ node contains the chain of its parent as a subsequence. We say a
1930 chain is *safe* if it has at most one ring- ϕ node. We will maintain the safe invariant for the net tree.

1931 **Invariant 1 (Safe invariant).** *The chain of every node in the net tree is safe.*

1932 If a chain of (t, i) is safe and contains (exactly one) ring- ϕ node, we will keep track of this node at
1933 (t, i) as its obligation.

1934 **Definition 16 (Obligation).** *Given a node (t, i) , the **obligation** of (t, i) , denoted by $\text{OBLIGATE}(t, i)$, is the
1935 lowest ring- ϕ node in its (safe) chain. If there is no ring- ϕ node in the chain, then $\text{OBLIGATE}(t, i) = \text{null}$.*

INSERT(q, T):

1. Let $u \leftarrow \text{CONTAINMENTSEARCH}(q, T)$ $\ll u$ is a candidate parent of $q \gg$
2. Node u could be the parent of q , unless in the following two cases:
 - (a) If u is the top of a jump and q is in ring α, β , or ϕ of u , then the parent of q is a (possibly hidden) node in the jump starting at u . We invoke $u \leftarrow \text{FINDPARENTINJUMP}(q, u, T)$ to find q 's parent.
 - (b) If q is completely out of ring ϕ of u (and u does not have to be the top of a jump), then we need to find another node to be the parent of p , by invoking $u \leftarrow \text{FINDPARENT}(u, T)$.
3. Let $u = (t', i')$. We create a node $(q, i' - 1)$ as a child of (t', i') by invoking $\text{ADD}(u, q, T)$. Then we invoke JUMPCREATE to create a jump from $(q, i' - 1)$ down to $(q, 0)$.
4. If q is in ring ϕ of u and $\text{OBLIGATE}(u) \neq \text{null}$, we invoke $\text{CHAINFIX}(u, T)$. This step modifies the tree to guarantee that all chains are safe after we add $(q, i' - 1)$ as a child of u in Step 3.
5. For every new node (x, j) that we create from Step 1 to Step 4, we check (and fix) the jump isolation of jumps nearby (x, j) by invoking $\text{MAINTAINJUMPISOLATION}(x, j, T)$. (There are only $O(1)$ such nodes.)

Figure 16: The insert procedure.

1936 Figure 16 describes the pseudo-code of the insertion operation. It might call two helper procedures
1937 $\text{FINDPARENTINJUMP}(q, u, T)$ and $\text{FINDPARENT}(q, u, T)$ described in Figure 17. The former finds a parent
1938 for q in the jump starting at a node u , while the latter finds a parent for q by checking nearby nodes of u .
1939 The following observation is immediate from the construction.

1940 **Observation 8.** *Given a new point q , let $u = (t, i)$ be the result of $\text{CONTAINMENTSEARCH}(q, T)$. In the
1941 end of $\text{INSERT}(q, T)$, we create $(q, i' - 1)$ as a child of an existing node (t', i') in the tree where $i' \leq i + 1$.*

1942 **Claim 3.** *In step 1 of $\text{FINDPARENT}(q, u, T)$, we consider nodes within $2\frac{\delta}{\varepsilon^{i+1}}$ to the parent of $u = (t, i)$ and
1943 choose $(v, i + 1)$ closest to q . Then $(v, i + 1)$ is closest to q among nodes at level $i + 1$.*

1944 **Proof:** Let $(t', i + 1)$ be the parent of (t, i) ; if (t, i) is the bottom of a jump, then $(t', i + 1) = (t, i + 1)$. By
1945 the covering property, $d_X(t, t') \leq \phi \frac{\delta}{\varepsilon^{i+1}}$. Recall that $u = (t, i)$ is the result of $\text{CONTAINMENTSEARCH}(q, T)$,
1946 thus $d_X(q, t) \leq \psi \frac{\delta}{\varepsilon^i}$. By triangle inequality, we have:

$$\begin{aligned} d_X(t', q) &\leq d_X(t', t) + d_X(q, t) \leq \phi \frac{\delta}{\varepsilon^{i+1}} + \psi \frac{\delta}{\varepsilon^i} \\ &= (\phi + \varepsilon\psi) \frac{\delta}{\varepsilon^{i+1}} \leq \frac{\delta}{\varepsilon^{i+1}} \quad (\text{since } \psi = \frac{5}{4}, \phi = \frac{3}{4} \text{ and } \varepsilon \leq \frac{1}{8}) \end{aligned}$$

FINDPARENTINJUMP(q, u, T):

Let $u = (t, i)$. We only invoke this procedure when u is the top of a jump, and q is in ring α, β or ϕ of u . Let (t, j) be the bottom of the jump at u .

1. Let $k \in (j, i]$ be such that $\alpha \frac{\delta}{\varepsilon^{k-1}} < d_X(t, q) \leq \alpha \frac{\delta}{\varepsilon^k}$.
2. Invoke JUMPSPLIT(t, i, j, k, T) to create (t, k) and (possibly) $(t, k-1)$ in the middle of the jump.
3. Return (t, k) .

FINDPARENT(q, u, T): *find a node at level $i+1$ to be the parent of q*

Let $u = (t, i)$. We only invoke this procedure when q is completely out of ring ϕ of u . To maintain the covering property, we guarantee that t is not completely out of ring ϕ of its parent.

1. Consider the nodes within the distance $2 \frac{\delta}{\varepsilon^{i+1}}$ to the parent of (t, i) ; the parent has pointers to all these nodes, and there are only $O_\lambda(1)$ of them. Let $(v, i+1)$ be the node closest to q among them. By Claim 3, $(v, i+1)$ is closest to t among nodes at level $i+1$.
A corner case is when (t, i) is the bottom of a jump, and hence, $(t, i+1)$ is a hidden node. Any node $(y, i+1)$ has $d_X(y, t) > 2 \frac{\delta}{\varepsilon^{i+1}}$. Thus, $(t, i+1)$ is closest to q , we create $(t, i+1)$ by using JUMPSPLIT.
2. Now we consider q and rings of $(v, i+1)$:
 - (a) If q is in ring α, β or ϕ of $(v, i+1)$: return $(v, i+1)$.
 - (b) If q is completely out of ring ϕ of $(v, i+1)$: We call PROMOTE(t, i, T) to promote (t, i) and return the parent of (t, i) . We will show in Claim 4 that (t, i) is a ring- ϕ node.

CHAINFIX(u, T)

Let $u = (t, i)$, if q satisfies $d_X(t, q) \in (\beta \frac{\delta}{\varepsilon^i}, \phi \frac{\delta}{\varepsilon^i}]$, we promote OBLIGATE(u) if it is not null.

Figure 17: Find parent for a new point q given access to a node u , and fix a chain at a node u .

If $(v, i+1)$ is closest to q among nodes at level $i+1$, then $d_X(v, q) \leq d_X(t', q) = \frac{\delta}{\varepsilon^{i+1}}$. This implies $d_X(v, t') \leq d_X(v, q) + d_X(q, t') \leq 2 \frac{\delta}{\varepsilon^{i+1}}$. Thus considering nodes within a distance $2 \frac{\delta}{\varepsilon^{i+1}}$ to $(t', i+1)$ suffices to find $(v, i+1)$. \square

In step 2(b) of FINDPARENT(q, u, T), we promote (t, i, T) when q is completely of ring ϕ of $(v, i+1)$. As promote could only be applied to ring- ϕ node, we show below that (t, i) is a ring- ϕ node.

Claim 4. *Node (t, i) in step 2(b) of FINDPARENT(q, u, T) is a ring- ϕ node.*

Proof: For contradiction, suppose that (t, i) is not a ring- ϕ node. By the covering property, it must be either a ring- α or ring- β node. Let $(t', i+1)$ be the parent of (t, i) ; if (t, i) is the bottom of a jump, then let $(t', i+1) = (t, i+1)$. We have $d_X(t, t') \leq \beta \frac{\delta}{\varepsilon^{i+1}}$. Since $u = (t, i)$ is the result of CONTAINMENTSEARCH(q, T), by Equation (7), we have $d_X(t, q) \leq \psi \frac{\delta}{\varepsilon^i}$. Recall that in step 2 of FINDPARENT, $(v, i+1)$ is closest to q among nodes at level $i+1$ and hence, $d_X(v, q) \leq d_X(t', q)$. We have:

$$\begin{aligned} d_X(v, q) &\leq d_X(t', q) \leq d_X(t, q) + d_X(t, t') \\ &\leq \psi \frac{\delta}{\varepsilon^i} + \beta \frac{\delta}{\varepsilon^{i+1}} \leq \phi \frac{\delta}{\varepsilon^{i+1}} \quad (\text{since } \varepsilon\psi + \beta \leq \phi \text{ in Equation (6)}), \end{aligned}$$

1960 implying that q is in ring α, β or ϕ of $(v, i+1)$, and hence step 2(b) will not be invoked, a contradiction. \square

1961 We remarked earlier that chains and obligations help avoid a cascading sequence of promotions. This
1962 is because when we promote a ring- ϕ node (t, i) , it becomes a ring- α node (Observation 7). However,
1963 its parent, which is $(t, i+1)$, might still be ring- ϕ . By the chain safe invariant (Invariant 1), the parent
1964 of $(t, i+1)$ is not a ring- ϕ node, and hence further promotion is needed (to guarantee the covering
1965 property).

1966 **About maintaining obligations.** For simplicity of the presentation, we do not explicitly include
1967 obligation maintenance in the above pseudocodes. We use the (simple yet clever) idea of Cole and
1968 Gottlieb [12], which we now briefly describe. Every node has a pointer to its obligation: a ring- α points
1969 to null, a ring- ϕ node points to itself, and a ring- β node points to the same value to which its parent
1970 points. (Specifically, if the parent of a ring- β node is another ring- β node, then they point to the same
1971 obligation, which could be a ring- ϕ node or null.) So, the focus is on maintaining the obligation of a
1972 ring- β node.

1973 Think about a ring- β node (t, i) that points to an ancestor ring- ϕ node (x, j) for some level $j > i$.
1974 Then all the nodes between (t, i) and (x, j) —except (x, j) —are ring- β nodes. The key observation is
1975 that when a new point q is inserted, it will never be inserted as an intermediate node between (t, i) and
1976 (x, j) , so the pointers of these nodes do not change. But it is possible that (x, j) will be promoted due to
1977 CHAINFIX and hence (x, j) is no longer a ring- ϕ node. However, by Observation 7, after the promotion,
1978 (x, j) becomes a ring- α node, and hence its obligation is null. By setting $\text{OBLIGATE}(x, j)$ to null, the
1979 obligation pointers of all the nodes from (x, j) down to (t, i) are also automatically set to null since they
1980 all point to $\text{OBLIGATE}(x, j)$. Therefore, maintaining obligations only adds $O(1)$ overhead.

1981 6.4 Analysis

1982 **Space and time.** We use the containment search data structure of Cole and Gottlieb [12] and modify
1983 INSERT with $O(1)$ steps, thus our data structure takes $O(n)$ space and the running time is $O_\lambda(\log(n))$ for
1984 each search, insertion, deletion, as claimed in Theorem 10.

1985 By Claim 2, given (x, i) is not a bottom or a hidden node of a jump, we can find (y, i) where
1986 $d_X(x, y) \leq g \cdot \frac{\delta}{\epsilon^i}$ in $O(g)^\lambda$ time.

1987 Next, we focus on showing packing, covering, close-containment, and b -jump isolation properties
1988 by induction. Specifically, we assume that these properties hold before an update to the net tree, and
1989 we will show them after the update. Deleting a point is simply marking the corresponding leaf of that
1990 point as deleted, and hence, none of the properties will be violated after a deletion. The difficult case
1991 is insertions, which involve creating new nodes and updating the parents of existing nodes in the tree,
1992 potentially violating the net tree properties.

1993 **Packing property.** Before showing the packing property, we give some simple observations.

1994 **Observation 9.** Given a jump from (t, i) down to (t, j) , if (t, k) is a hidden node or (t, k) is created by
1995 operations JUMPSPLIT or JUMPFIX for some level $k \in (i, j)$, then $d_X(t, v) > b \frac{\delta}{\epsilon^k}$ for every node (v, k) at
1996 the same level k .

1997 **Proof:** By the definition of a jump, any node (v, k) for $v \neq t$ and $k \in (j, i)$ is not a descendant of (t, i) .
1998 By b -jump isolation property, $d_X(t, v) > b \frac{\delta}{\epsilon^k}$. \square

1999 **Observation 10.** PROMOTE maintains the packing property.

2000 **Proof:** PROMOTE creates at most one new node, and it checks the packing property in Step 1: before
 2001 create $(t, i + 1)$ as a new parent of (t, i) , if there exists a node $(v, i + 1)$ such that $d_X(v, t) \leq \alpha \frac{\delta}{\varepsilon^i}$, then it
 2002 changes the parent of (t, i) to $(v, i + 1)$ and terminates. Thus, the packing property is maintained. \square

2003 **Lemma 17 (Packing property).** *For any pair of nodes (x, i) and (y, i) at level i , $d_X(x, y) > \alpha \frac{\delta}{\varepsilon^i}$ for
 2004 $\alpha = 1/4$.*

2005 **Proof:** It suffices to show that when a new point is added, creating new nodes does not violate the
 2006 packing property. There are three cases where new nodes are created by an insertion: (i) nodes created
 2007 by PROMOTE (called in Step 2 case (b) and Step 4 of INSERT), (ii) nodes created by JUMPSPLIT (called
 2008 in Step 2 cases (a) and (b) of INSERT), or JUMPFIX (possibly invoked in Step 5 of INSERT), (iii) node
 2009 $(q, i' - 1)$ for a new point q created by Step 3 of INSERT for $i' > 0$.

2010 Observation 10 takes care of case (i). For (ii), by Observation 9, a node at level k created by JUMPFIX
 2011 or JUMPSPLIT is at distance $b \frac{\delta}{\varepsilon^k}$ to any existing node at level k , which implies the packing property since
 2012 $b > \alpha$ ($b \geq \frac{5}{4}$, $\alpha = \frac{1}{4}$). Both cases (i) and (ii) imply that Steps 2, 4, and 5 of INSERT maintain the packing
 2013 property.

2014 It remains to consider case (iii). Suppose that there exists a node $(x, i' - 1)$ at the same level with
 2015 $(q, i' - 1)$ such that $d_X(x, q) \leq \alpha \frac{\delta}{\varepsilon^{i'-1}}$. Let u be the result of $\text{CONTAINMENTSEARCH}(q, T)$. We claim that:

$$u = (t, i) \text{ has the same level as } (q, i' - 1); \text{ that is, } i = i' - 1. \quad (8)$$

2017 To see (8), recall that the containment search finds the lowest node $u = (t, i)$ where $d_X(t, q) \leq \psi \frac{\delta}{\varepsilon^i}$,
 2018 for $\psi = \frac{5}{4}$. Since $\alpha = \frac{1}{4} < \psi$, $d_X(x, q) < \psi \frac{\delta}{\varepsilon^{i'-1}}$, implying that $i \leq i' - 1$. By Observation 8, the parent of
 2019 q is a node at a level at most $i + 1$, giving $i' \leq i + 1$ and hence $i \geq i' - 1$. We conclude that $i' = i + 1$ as
 2020 claimed in (8).

2021 Since u has the same level as $(q, i' - 1)$, it cannot be q 's parent or ancestor. Observe that only in Step
 2022 2 of $\text{INSERT}(q, T)$ we might find a parent for q different from u . We consider two cases:

- 2023 • **Step 2(a):** In this case, u is the top of a jump and q is in ring α, β or ϕ of u . Then, the parent of q
 2024 is a descendant of u , contradicting (8).
- 2025 • **Step 2(b):** In this case, q is completely out of ring ϕ of u and hence $d_X(t, q) > \phi \frac{\delta}{\varepsilon^{i'-1}}$. By the
 2026 definition of containment search, u is the node closest to q at level $i' - 1$. Thus, $d_X(x', q) \geq$
 2027 $d_X(t, q) > \phi \frac{\delta}{\varepsilon^{i'-1}}$, contradicting the assumption that $d_X(x, q) \leq \alpha \frac{\delta}{\varepsilon^{i'-1}}$.

2028 Both cases above imply the packing property. \square

2029 **Covering property.** The proof is similar to the packing property. We observe that:

2030 **Observation 11.** *Given a node ring- ϕ node (t, i) , if the safe invariant (Invariant 1) and the covering
 2031 property are maintained, then the parent of (t, i) is either a ring- α or ring- β node.*

2032 **Proof:** By Observation 5 item (c), the parent of (t, i) is a node at level $i + 1$, denoted by $(u, i + 1)$.
 2033 Consider the safe chain of (t, i) , starting from the lowest ring- α ancestor of (t, i) and ending at (t, i) .
 2034 Since the chain is safe, it has at most one ring- ϕ node, and in this case, must be (t, i) . Therefore, no
 2035 other node in the chain, which includes $(u, i + 1)$, is a ring- ϕ node. By the covering property, $(u, i + 1)$
 2036 must either be a ring- α or ring- β node. \square

2037 **Claim 5.** PROMOTE maintains the covering property.

2038 **Proof:** Recall that $\text{PROMOTE}(t, i)$ updates the parent of (t, i) and possibly creates one new node $(t, i+1)$
 2039 as the parent of (t, i) . By Observation 7, after $\text{PROMOTE}(t, i, T)$, (t, i) becomes a ring- α node, which
 2040 implies the covering property for (t, i) .

2041 It remains to consider the case where PROMOTE creates $(t, i+1)$, and we have to show the covering
 2042 property for $(t, i+1)$. Let $(u, i+1)$ be the parent of (t, i) before $\text{PROMOTE}(t, i, T)$, $(v, i+2)$ be the
 2043 parent of $(u, i+1)$, and $(v', i+2)$ be the node at level $i+2$ that is closest to t . By definition of
 2044 $(v', i+2)$, $d_X(t, v') \leq d_X(t, v)$. In step 3 of PROMOTE , we choose either $(v, i+2)$ or $(v', i+2)$ to be the
 2045 parent of $(t, i+1)$. Recall that (t, i) must be a ring- ϕ node as this is the condition to invoke PROMOTE ,
 2046 giving $d_X(t, u) \leq \phi \frac{\delta}{\varepsilon^{i+1}}$. In addition, by Observation 11, $(u, i+1)$ is a ring- α or ring- β node, giving
 2047 $d_X(u, v) \leq \beta \frac{\delta}{\varepsilon^{i+2}}$. By triangle inequality, we obtain:

$$\begin{aligned} d_X(t, v') &\leq d_X(t, v) \leq d_X(t, u) + d_X(u, v) \leq \phi \frac{\delta}{\varepsilon^{i+1}} + \beta \frac{\delta}{\varepsilon^{i+2}} \\ &\leq \phi \frac{\delta}{\varepsilon^{i+2}} \quad (\text{since } \beta = \frac{2}{4}, \phi = \frac{3}{4}, \varepsilon \leq \frac{1}{8}) \end{aligned}$$

2049 Therefore, the covering property holds for $(t, i+1)$. \square

2050 Now, we are ready to show the covering property for all nodes in the tree.

2051 **Lemma 18 (Covering property).** *If (y, i) has a child (x, i') for $i' < i$ then $d_X(x, y) \leq \phi \frac{\delta}{\varepsilon^i}$ with $\phi = \frac{3}{4}$.*

2052 **Proof:** It suffices to focus on the edges of the net tree, which are changed or added by INSERT since
 2053 deletions are only marked. Two types of update: (i) modifying existing points in the tree, involving
 2054 changing parents, or creating new nodes and edges for existing points., (ii) finding the parent for a
 2055 newly inserted point.

2056 In case (i), the tree is modified by one of the following internal operations: (1) PROMOTE , or (2)
 2057 JUMPFIX or JUMPSPLIT . Claim 5 takes care of (1). For (2), by the definition of a jump, the new node in
 2058 the middle of a jump is a ring- α node, thus the covering property is maintained.

2059 For case (ii), let $u = (t, i)$ be the result of $\text{CONTAINMENTSEARCH}(q, T)$. If step 2 of INSERT is not
 2060 applied, then q is in ring α, β or ϕ of u and u is not the top of a jump. The covering property follows
 2061 from the fact that $(q, i-1)$ as a child of $u = (t, i)$. Thus, it remains to consider step 2.

2062 In step 2(a), $u = (t, i)$ is the top of a jump and q is in ring α, β or ϕ of u . The parent of q is a node
 2063 (t, k) in the jump from (t, i) down to (t, j) for some $k \in (j, i]$. If $k = i$, since q is in ring α, β or ϕ of
 2064 $u = (t, i)$, the covering property holds. Otherwise, $k < i$. By the definition of k , $d_X(t, q) \leq \alpha \frac{\delta}{\varepsilon^k}$ and since
 2065 $\alpha < \phi$ ($\alpha = \frac{1}{4}, \phi = \frac{3}{4}$), the covering property holds.

2066 In step 2(b), we invoke FINDPARENT to find a node at level $i+1$ to be the parent of q . In step 2(a)
 2067 of FINDPARENT , the parent $(v, i+1)$ of q has the property that q is in its ring α, β or ϕ , and hence the
 2068 covering property holds. In step 2(b) of FINDPARENT , q is completely out of ring ϕ of $(v, i+1)$, we
 2069 promote (t, i) and choose the parent of (t, i) to be the parent of q . By Claim 4, (t, i) is a ring- ϕ node. Let
 2070 $(t'', i+1)$ be the new parent of (t, i) after $\text{PROMOTE}(t, i)$; $(t'', i+1)$ is also the parent of q by construction
 2071 in step 2(b). By Observation 7, (t, i) becomes a ring- α node, which means $d_X(t, t'') \leq \alpha \frac{\delta}{\varepsilon^{i+1}}$. We have:

$$\begin{aligned} d_X(t'', q) &\leq d_X(t'', t) + d_X(t, q) \leq \alpha \frac{\delta}{\varepsilon^{i+1}} + \psi \frac{\delta}{\varepsilon^i} \\ &\leq \beta \frac{\delta}{\varepsilon^{i+1}} \quad (\text{since } \alpha + \varepsilon \psi \leq \beta \text{ in Equation (6)}) \\ &< \phi \frac{\delta}{\varepsilon^{i+1}}, \end{aligned}$$

2073 the covering property holds. \square

2074 **Close-containment property.** Close-containment property follows directly from the covering property.

2075 **Lemma 19 (Close-containment).** *If (x, i) has a descendant (y, k) then $d_X(x, y) \leq \gamma \frac{\delta}{\varepsilon^i} - \gamma \frac{\delta}{\varepsilon^k}$.*

2076 **Proof:** Let (y, k) be a descendant of (x, i) for $k < i$. By Lemma 18, we have $d_X(x, y) \leq \phi \frac{\delta}{\varepsilon^i} + \phi \frac{\delta}{\varepsilon^{i-1}} +$
2077 $\dots + \phi \frac{\delta}{\varepsilon^{k+1}}$. If $k = i - 1$, then:

$$2078 d_X(x, y) \leq \phi \frac{\delta}{\varepsilon^i} \leq \gamma \frac{\delta}{\varepsilon^i} - \gamma \frac{\delta}{\varepsilon^{i-1}}.$$

2079 The last inequality holds since $\phi \leq \gamma(1 - \varepsilon)$ for $\phi = \frac{3}{4}, \gamma = 1, \varepsilon \leq \frac{1}{8}$. By induction:

$$2080 \begin{aligned} d_X(x, y) &\leq \phi \frac{\delta}{\varepsilon^i} + \dots + \phi \frac{\delta}{\varepsilon^{k+1}} + \phi \frac{\delta}{\varepsilon^{k+1}} \\ &\leq \left(\gamma \frac{\delta}{\varepsilon^i} - \gamma \frac{\delta}{\varepsilon^{i-1}} \right) + \left(\gamma \frac{\delta}{\varepsilon^{i-1}} - \gamma \frac{\delta}{\varepsilon^{i-2}} \right) \dots \left(\gamma \frac{\delta}{\varepsilon^{k+1}} - \gamma \frac{\delta}{\varepsilon^k} \right) \\ &= \gamma \frac{\delta}{\varepsilon^i} - \gamma \frac{\delta}{\varepsilon^k}, \end{aligned}$$

2081 as desired. \square

2082 **Safe chain invariant.** Recall that the chain of a node (t, i) is the sequence of nodes starting from the
2083 lowest ring- α ancestor of (t, i) , denoted by (t^*, i^*) , to (t, i) . It is safe if it contains at most one ring- ϕ
2084 node. Recall that the obligation of (t, i) is the ring- ϕ node in its (safe) chain. Directly from the definition:

2085 **Observation 12.** (a) *The chain of a ring- α node is safe.*

2086 (b) *The chain of a ring- β node is safe if the chain of its parent is safe. The obligation of a ring- β node
2087 is the obligation of its parent.*

2088 (c) *The chain of a ring- ϕ node is safe if its parent has a null obligation.*

2089 We now show the safe invariant.

2090 **Lemma 20 (Safe invariant).** *The chain of every node is always safe.*

2091 **Proof:** Assume that all chains are safe before an update to the net tree; we show that they remain safe
2092 after an update. After step 2 of INSERT, we found a node $u = (t, i)$ to be a parent of q such that q is not
2093 completely out of ring ϕ of u . (We use the notation (t, i) for u instead of (t', i') as in step 2 to avoid
2094 clutter.) If (a) q is in ring α or β of (t, i) or (b) q is in ring ϕ of (t, i) and the chain of (t, i) has no
2095 obligation, then adding $(q, i-1)$ as a child of (t, i) does not violate the safe invariant. The remaining
2096 case is when q is in ring ϕ of (t, i) and the chain of (t, i) has a ring- ϕ node. In this case, step 4 of INSERT
2097 promotes the obligation of (t, i) . Let $(t_j, j) = \text{OBLIGATE}(t, i)$. We have to show two things:

- 2098 (i) after promoting (t_j, j) , we have to show that the chain of (t, i) has no ring- ϕ node, thus we can
2099 add $(q, i-1)$ as a ring- ϕ child of (t, i) .
- 2100 (ii) Since promoting (t_j, j) changes the parent of (t_j, j) and possibly creates a new node $(t_j, j+1)$, we
2101 also have to show that the chains of (t_j, j) and $(t_j, j+1)$ are safe.

2102 We first focus on (i). Before $\text{PROMOTE}(t_j, j, T)$, the chain of (t, i) starts from the lowest ring- α ancestor
2103 of (t, i) , say (t^*, i^*) . At this point, the sequence of nodes from (t^*, i^*) to (t, i) has only one ring- α node,
2104 which is (t^*, i^*) , and only one ring- ϕ node, which is (t_j, j) . After $\text{PROMOTE}(t_j, j, T)$, (t_j, j) becomes a

2105 ring- α node by Observation 7 and therefore, the lowest ring- α ancestor of (t, i) . Thus, the chain of (t, i)
 2106 starts from (t_j, j) , and this chain has no ring- ϕ node, as desired.

2107 For (ii), as (t_j, j) becomes a ring- α node after after PROMOTE(t_j, j, T), its chain is safe by Observa-
 2108 tion 12. We now focus on the chain of $(t_j, j + 1)$. If $(t_j, j + 1)$ is a ring- α or ring- β node, then its chain is
 2109 safe by induction. The remaining case is when $(t_j, j + 1)$ is a ring- ϕ node. We first claim that:

2110 **Claim 6.** *Before PROMOTE(t_j, j, T), let (t^*, i^*) be the lowest ring- α ancestor of (t, i) . After PROMOTE(t_j, j, T),
 2111 if $(t_j, j + 1)$ is a ring- ϕ node, then the chain of $(t_j, j + 1)$ starts at (t^*, i^*) .*

2112 To see the claim, let $(x, j + 1)$ be the parent of (t_j, j) before PROMOTE(t_j, j, T). Since $(t_j, j + 1)$ is created,
 2113 step 3 of PROMOTE will find a parent for $(t_j, j + 1)$. Since $(t_j, j + 1)$ is a ring- ϕ , by the construction of step
 2114 3, the parent of $(x, j + 1)$, denoted by $(v, j + 2)$, will be chosen as the parent of $(t_j, j + 1)$. Furthermore,
 2115 $(x, j + 1)$ cannot be a ring- α node since otherwise, by triangle inequality:

$$\begin{aligned} d_X(v, t_j) &\leq d_X(v, x) + d_X(x, t_j) \\ &\leq \alpha \frac{\delta}{\varepsilon^{j+2}} + \phi \frac{\delta}{\varepsilon^{j+1}} \quad (\text{since } (x, j + 1) \text{ is a ring-}\alpha \text{ node}) \\ &\leq \beta \frac{\delta}{\varepsilon^{j+2}} \quad (\text{since } \alpha + \phi \varepsilon \leq \beta \text{ by Equation (6)}) , \end{aligned}$$

2117 contradicting that $(t_j, j + 1)$ is a ring- ϕ node. As $(x, j + 1)$ and (t_j, j) are not ring- α node, (t^*, i^*) is also
 2118 the lowest ring- α ancestor of $(v, j + 2)$; it could be that $(t^*, i^*) = (v, j + 2)$. Since $(t_j, j + 1)$ is not a ring- α
 2119 node and has $(v, j + 2)$ as the parent, (t^*, i^*) is also the lowest ring- α ancestor of $(t_j, j + 1)$, implying
 2120 Claim 6.

2121 Observe that before the promotion of (t_j, j) , the path from (t^*, i^*) to (t_j, j) has only one ring- ϕ node,
 2122 which is (t_j, j) . As (t^*, i^*) is an ancestor of $(t_j, j + 1)$ after the promotion of (t_j, j) by Claim 6, there is
 2123 no ring- ϕ node from (t^*, i^*) to the parent of $(t_j, j + 1)$. Thus, even if $(t_j, j + 1)$ becomes a ring- ϕ node,
 2124 the chain of $(t_j, j + 1)$ is still safe. \square

2125 **Jump isolation property.** This is the last property that we have to show to complete the proof of
 2126 Theorem 10. In INSERT, after we add $O(1)$ new nodes and modify existing nodes from step 1 to step 4,
 2127 we check and fix the jump isolation property in step 5 by invoking MAINTAINJUMPISSOLATION(y, i, T) for
 2128 every new node (y, i) . While the jump isolation property for a jump from (x, i) down to (x, j) is defined
 2129 w.r.t every node at level $k \leq i$, the checking procedure MAINTAINJUMPISSOLATION only looks at top of the
 2130 jump (and hence the checking and fixing can be done in $O_\lambda(1)$ time), which is justified by the following
 2131 lemma.

2132 **Lemma 21.** *Given $\varepsilon \leq \frac{a}{b}$ and a jump (x_0, i_0) down to (x_0, j_0) , if $d_X(x_0, v_0) > \alpha \frac{\delta}{\varepsilon^{i_0}} + \gamma \frac{\delta}{\varepsilon^{i_0}}$ for every other
 2133 node (v_0, i_0) at level i_0 , then the jump starting at (x_0, i_0) is b -isolated: any non-descendant node (z, m)
 2134 of (x_0, i_0) for $m < i_0$ has $d_X(x_0, z) > b \frac{\delta}{\varepsilon^m}$. (This includes the case where (z, m) is a newly created node.)*

2135 **Proof:** First, we consider when (z, m) does not have an ancestor at level i_0 . In this case, there must be a
 2136 jump J_t from a node (t, i') to (t, j') where $j' < i_0 < i'$ such that (t, j') is an ancestor of (z, m) . Observe
 2137 that (x_0, i_0) is not a descendant of (t, i') . By induction, J_t is b -isolated, implying that $d_X(x_0, t) > b \frac{\delta}{\varepsilon^{i_0}}$.
 2138 Since (z, m) is a descendant of (t, j') , by the close-containment property, we have $d_X(z, t) \leq \gamma \frac{\delta}{\varepsilon^{j'}} - \gamma \frac{\delta}{\varepsilon^m} \leq$
 2139 $\gamma \frac{\delta}{\varepsilon^{i_0-1}} - \gamma \frac{\delta}{\varepsilon^m}$. By triangle inequality:

$$\begin{aligned} d_X(x_0, z) &\geq d_X(x_0, t) - d_X(t, z) \\ &> b \frac{\delta}{\varepsilon^{i_0}} - (\gamma \frac{\delta}{\varepsilon^{i_0-1}} - \gamma \frac{\delta}{\varepsilon^m}) \\ &\geq (b - \gamma \varepsilon) \frac{\delta}{\varepsilon^{i_0}} = \frac{(b - \gamma \varepsilon)}{\varepsilon} \frac{\delta}{\varepsilon^{i_0-1}} \geq b \frac{\delta}{\varepsilon^m} \end{aligned}$$

where the last inequality holds since $m \leq i_0 - 1$, $\alpha = \frac{1}{4}$, $\gamma = 1$, $b \geq \frac{5}{4}$, and $\varepsilon \leq \frac{\alpha}{b} \leq \frac{b}{b+\gamma}$.

Now, we consider the case where (z, m) has an ancestor at level i_0 , let this node be (v_0, i_0) . For contradiction, suppose that $d_X(z, m) \leq b \frac{\delta}{\varepsilon^m}$. By the close-containment property, we have $d_X(z, v_0) \leq \gamma \frac{\delta}{\varepsilon^{i_0}} - \gamma \frac{\delta}{\varepsilon^m}$. Thus:

$$\begin{aligned} d_X(x_0, v_0) &\leq d_X(x_0, z) + d_X(z, v_0) \\ &\leq b \frac{\delta}{\varepsilon^m} + (\gamma \frac{\delta}{\varepsilon^{i_0}} - \gamma \frac{\delta}{\varepsilon^m}) \leq \alpha \frac{\delta}{\varepsilon^{i_0}} + \gamma \frac{\delta}{\varepsilon^{i_0}} \end{aligned}$$

where $b \frac{\delta}{\varepsilon^m} \leq \alpha \frac{\delta}{\varepsilon^{i_0}}$ holds since $m \leq i_0 - 1$ and $\varepsilon \leq \frac{\alpha}{b}$. Thus if $d_X(x_0, v_0) > \alpha \frac{\delta}{\varepsilon^{i_0}} + \gamma \frac{\delta}{\varepsilon^{i_0}}$, then $d_X(x_0, z) > b \frac{\delta}{\varepsilon^m}$. \square

We are now ready to show the b -jump isolation property.

Lemma 22 (b-Jump Isolation). *Every jump is b -isolated.*

Proof: Recall that in Step 5 of `INSERT`, we invoke `MAINTAINJUMPIISOLATION`(y, i, T) for any new node (y, i) . In this operation, we find (x, i) where $d_X(x, y) \leq (\alpha + \gamma) \frac{\delta}{\varepsilon^i}$.

If (x, i) does not exist, `MAINTAINJUMPIISOLATION` does nothing. If (y, i) is the top of a jump J_y , then J_y is b -isolated by Lemma 21. On the other hand, the b -jump isolation property is maintained for every existing jump J_u starting at (u, i') . Specifically, if $i' > i$, adding a node at level i does not violate the b -isolation property of J_u by applying Lemma 21 with $i_0 = i'$ (and $(z, m) = (y, i)$). If $i' = i$, by induction J_u is b -isolated with respect to existing nodes before adding (y, i) . When adding (y, i) , since $d_X(u, y) > (\alpha + \gamma) \frac{\delta}{\varepsilon^i}$, by Lemma 21, the b -jump isolation property at J_u is maintained. When $i' < i$, adding a node at level i does not change anything at level i' , and hence Lemma 21 also applies here. In all cases, the construction maintains the b -jump isolation property.

We now consider the complementary case where there exists a node (x, i) such that $d_X(x, y) \leq (\alpha + \gamma) \frac{\delta}{\varepsilon^{i-1}}$. If (x, i) (or (y, i)) is the top of a jump J_x (or J_y), in `MAINTAINJUMPIISOLATION`(y, i, T), we invoke `JUMPFIX` to create $(x, i-1)$ (or $(y, i-1)$). Node (x, i) and (y, i) are no longer the top of their jumps, and hence J_x and J_y are effectively replaced by two new jumps, denoted by J'_x and J'_y , starting at $(x, i-1)$ and $(y, i-1)$, respectively. And we need to argue that after adding these new jumps, the jump isolation property is fixed.

Let us consider J'_y first. We claim that:

$$\text{for } J'_y \text{ to exist, } y \text{ must be a newly inserted point.} \quad (9)$$

For contradiction, suppose that y is an existing point. Then (y, i) is a new node created by `PROMOTE` or `JUMPSPLIT` in step 2 or step 4 of `INSERT`(q, t) for some point $q \neq y$. If (y, i) is created by promoting $(y, i-1)$, then $(y, i-1)$ exists and there is no jump starting from (y, i) to fix. In the other case, (y, i) is created by splitting a jump from (y, i'') down to (y, j'') at level i where $i'' > i$. Since the jump starting at (y, i'') is b -isolated, by definition, $d_X(y, t) > b \frac{\delta}{\varepsilon^k}$ for any node (t, k) where $j'' < k < i''$. It follows that the jump starting at (y, i) , which is J_y , is also b -isolated, and hence `JUMPFIX` is not called on (y, i) . Therefore, (9) holds.

Since y is a newly inserted point, observe that any existing node $(v, i-1)$ must satisfy $d_X(v, y) > \psi \frac{\delta}{\varepsilon^{i-1}}$ since otherwise, `CONTAINMENTSEARCH`(y, T) will return a node at a level at most $i-1$, and hence after `INSERT`, the parent of y is a node at a level at most i by Observation 8. Furthermore, since $\psi \geq \alpha + \gamma$ by Equation (6), we have $d_X(v, y) > (\alpha + \gamma) \frac{\delta}{\varepsilon^{i-1}}$, implying the b -jump isolation of J'_y by Lemma 21.

Finally, we consider the jump J'_x . Before adding (y, i) , J_x satisfies the b -jump isolation. Adding a new node (y, i) does not change the distance from x to other nodes, and hence, the only possible jump violation to J'_x is due to $(y, i-1)$. Since (x, i) and (y, i) are two nodes at level i , by packing property, $d_X(x, y) > \alpha \frac{\delta}{\varepsilon^i}$, giving $d_X(x, y) > \psi \frac{\delta}{\varepsilon^{i-1}} \geq (\alpha + \gamma) \frac{\delta}{\varepsilon^{i-1}}$ since $\alpha \geq \psi \varepsilon \geq (\alpha + \gamma) \varepsilon$ is given by Equation (6). By Lemma 21, J'_x is b -isolated with respect to new node $(y, i-1)$, and hence J'_x is b -isolated overall. \square

2184 7 Applications of LSO

2185 Here we give the details of the applications of LSO mentioned in Section 1.2.

2186 7.1 Dynamic VFT Spanners

2187 **Theorem 4.** Given $\varepsilon \in (0, 1)$, $k \in [1, n - 2]$ and a dynamic point set S in doubling metrics of dimension
 2188 λ , there is a data structure \mathcal{D} such that \mathcal{D} (implicitly) maintains a $(k, 1 + \varepsilon)$ -VFTS H of degree $k \cdot \varepsilon^{-O(\lambda)}$
 2189 for S in $O(\log n \varepsilon^{-O(\lambda)})$ time per update, and \mathcal{D} returns all neighbours of a given vertex of H in $k \varepsilon^{-O(\lambda)}$
 2190 time. The update time and query time are optimal for fixed ε, λ .

2191 **Proof:** Our algorithm follows that of Chan, Har-Peled and Jones [10]. Statically, given a (τ, ε) -LSO Σ ,
 2192 we construct a k -VFTS H as follows:

2193 Initially, $H = (S, \emptyset)$. For each ordering $\sigma \in \Sigma$ and each point $q \in \sigma$, we add $2(k + 1)$ edges
 2194 incident to q to H where $k + 1$ edges are from q to its $k + 1$ nearest predecessors in σ and
 2195 the other $k + 1$ edges are from q to its $k + 1$ nearest successors in σ . (If q is close to the endpoints
 2196 of σ , then we might add less than $2(k + 1)$ edges.)

2197 The claim (which we will prove later) is that:

2198 **Claim 7.** H is a k -VFTS where every vertex has degree at most $\tau \cdot 2(k + 1) = O(\tau k)$.

2199 To maintain H dynamically, whenever a point q is added to S , we invoke $\text{INSERT}(q, \Sigma)$. Then, given
 2200 σ is the ordering i^{th} of Σ , we iteratively find $k + 1$ nearest predecessors $p_1 \prec_\sigma p_2 \prec_\sigma \dots \prec_\sigma p_{k+1}$ by
 2201 $p_j = \text{GETPREDECESSOR}(p_{j+1}, i, \Sigma)$ where $p_{k+2} = q$. Similarly, using GETSUCCESSOR , we find $k + 1$ nearest
 2202 successors of q : $s_1 \prec_\sigma s_2 \prec_\sigma \dots \prec_\sigma s_{k+1}$. For every $j \in [1, k + 1]$, we add to H edges (p_j, q) and (q, s_j) , then
 2203 remove from H the edge (p_j, s_j) .

2204 When a point q is deleted from S , first we reconnect neighbors of q in H as follows. For each ordering
 2205 $\sigma \in \Sigma$, get $k + 1$ nearest predecessors and $k + 1$ nearest successors of q in σ as described above. Let
 2206 $p_1 \prec_\sigma \dots \prec_\sigma p_{k+1}$ be $k + 1$ those predecessors, and $s_1 \prec_\sigma \dots \prec_\sigma s_{k+1}$ be $k + 1$ those successors. Add the
 2207 edge (p_j, s_j) to H for all $j \in [1, k + 1]$. Finally, remove all edges of q out of H and invoke $\text{DELETE}(q, \Sigma)$.

2208 By Theorem 1, INSERT and DELETE of Σ take $O(\log(1/\varepsilon)(\log n + \varepsilon^{-O(\lambda)}))$ time, while GETPREDECESSOR
 2209 and GETSUCCESSOR run in $O(1)$ per operation. Hence, the total time to add and remove edges regarding
 2210 an insertion or deletion is $O(\tau k) = k \varepsilon^{-O(\lambda)}$. In summary, the insertion and deletion time is in $(\log n + k) \varepsilon^{-O(\lambda)}$ as claimed in the theorem.

2211 To complete the proof of Theorem 4, we prove Claim 7. By Theorem 1, Σ is stable, thus it suffices to
 2212 get predecessors and successors at the point that is updated (inserted or deleted); all other edges remain
 2213 in H . By the construction, every vertex has a degree at most $\tau \cdot 2(k + 1)$. Now we show H is a k -VFTS of
 2214 S . Let F be the subset of S with size at most k . Consider two points s, t , there is an ordering $\sigma \in \Sigma$ such
 2215 that all points p between s, t have $d_X(p, s) \leq \varepsilon d_X(s, t)$ or $d_X(p, t) \leq \varepsilon d_X(s, t)$. Let σ' be the ordering
 2216 obtained from σ by removing points in F . Observe that among adjacent pairs in σ' , there are s', t' such
 2217 that (i) $s \preceq_{\sigma'} s' \prec_{\sigma'} t$ and $s \prec_{\sigma'} t' \preceq_{\sigma'} t$, (ii) $d_X(s, s') \leq \varepsilon d_X(s, t)$ and $d_X(t, t') \leq \varepsilon d_X(s, t)$. Since we add
 2218 edges of a point with its $k + 1$ predecessors and $k + 1$ successors to H , if u and v are adjacent in σ' , there
 2219 is an edge (u, v) in $H \setminus F$ and $d_{H \setminus F}(u, v) = d_X(u, v)$. It follows that $d_{H \setminus F}(s', t') = d_X(s', t')$. Now we prove
 2220 the claim by induction, and suppose that $d_{H \setminus F}(s, s') \leq (1 + c\varepsilon)d_X(s, s')$ and $d_{H \setminus F}(t, t') \leq (1 + c\varepsilon)d_X(t, t')$.

2222 By triangle inequality, we have:

$$\begin{aligned}
d_{H \setminus F}(s, t) &\leq d_{H \setminus F}(s, s') + d_{H \setminus F}(s', t') + d_{H \setminus F}(t', t) \\
&\leq (1 + c\epsilon)d_X(s, s') + d_X(s', t') + (1 + c\epsilon)d_X(t, t') \\
&\leq (1 + c\epsilon)(d_X(s, s') + d_X(t, t')) + (d_X(s', s) + d_X(s, t) + d_X(t, t')) \\
&\leq (1 + c\epsilon)2\epsilon d_X(s, t) + (d_X(s, t) + 2\epsilon d_X(s, t)) \\
&= (1 + 4\epsilon + 2c\epsilon^2)d_X(s, t) \\
&\leq (1 + c\epsilon)d_X(s, t)
\end{aligned}$$

2224 Setting $c = 8$, the inequality holds when $\epsilon \leq \frac{1}{4}$. By scaling ϵ with the constant factor $c = 8$, Claim 7
2225 holds. \square

2226 7.2 Dynamic Tree Cover

2227 **Theorem 5.** *Given a dynamic point set S in doubling metrics of dimension λ and any $\epsilon \in (0, 1)$, there is
2228 a data structure \mathcal{D}_J explicitly maintaining a tree cover \mathcal{J} for S such that J has stretch of $1 + \epsilon$ and size of
2229 $\epsilon^{-O(\lambda)}$, and the running time per update is $O(\epsilon^{-O(\lambda)} \log(n))$.*

2230 We can construct a tree cover \mathcal{J} from a pairwise tree cover \mathcal{T} by adding weights to edges:

- 2231 • [Step 1.] For every PIT $T \in \mathcal{T}$, we create a tree $J \in \mathcal{J}$ such that J and T has the same set of nodes
2232 and edges.
- 2233 • [Step 2.] For every edge connecting two nodes $a = (x, y, i)$ and $b = (u, v, j)$ in J , we assign a weight
2234 to the edge (a, b) of J as $w_J(a, b) = d_X(\{x, y\}, \{u, v\})$. (We use $d_X(A, B) = \min_{x \in A, y \in B} d_X(x, y)$ to
2235 denote the distance between two sets of points A and B .)
- 2236 • [Step 3.] We now update the weights of the edges from a node $a = (x, y, i)$ to its children. (Note
2237 that not all edges from a to its children get their weights updated.) Let $c = (s, t, i')$ be the parent
2238 of a .
 - 2239 – If $d_X(x, \{s, t\}) \leq d_X(y, \{s, t\})$, for every child $b = (u, v, j)$ of (x, y, i) such that $d_X(y, \{u, v\}) \leq$
2240 $d_X(x, \{u, v\})$ we add $d_X(x, y)$ to $w(a, b)$.
 - 2241 – If $d_X(x, \{s, t\}) > d_X(y, \{s, t\})$, for every child $b = (u, v, j)$ of (x, y, i) such that $d_X(x, \{u, v\}) <$
2242 $d_X(y, \{u, v\})$, we add $d_X(x, y)$ to $w(a, b)$.

2243 The intuition of the Step 3 is as follows. Suppose that $b = (u, v, j)$ is a child of a node $a = (x, y, i)$,
2244 we temporarily assign the weight of edge (a, b) to be the closest distance, namely $d_X(\{x, y\}, \{u, v\})$,
2245 between its labeled points in Step 2. One can think of this as using the closest pair to “represent” the
2246 edge (a, b) . Next, consider the parent $c = (s, t, i')$ of a ; in the same way, Step 2 also uses the closest
2247 pair in the labels of c and a to represent (a, c) . But this means the path from b to c (passing through a)
2248 might “miss” the edge (x, y) , and therefore, if $\{u, v\}$ is closer to x and $\{s, t\}$ is closer to y , we need to
2249 add $d_X(x, y)$ to the weight of edge (a, b) , as in Step 3.

2250 For a dynamic point set S , whenever we update \mathcal{T} , \mathcal{J} have the same updates as \mathcal{T} , and we assign
2251 weights to edges of trees in \mathcal{J} as Step 2 and Step 3 above. Note that \mathcal{J} and \mathcal{T} share many properties: the
2252 number of trees, update time, and the covering property. Since \mathcal{T} has $\epsilon^{-O(\lambda)}$ PITs, the tree cover \mathcal{J} also
2253 has $\epsilon^{-O(\lambda)}$ trees. The running time per update to S of \mathcal{T} is $O(\epsilon^{-O(\lambda)} \log n)$, thus \mathcal{J} has the same update
2254 time.

2255 It remains to show the stretch of \mathcal{J} . We rely on the pairwise covering property of \mathcal{T} for this: for any
2256 pair of points $x_0, y_0 \in S$ whose distance in $[\frac{\delta}{\epsilon^i}, \frac{2\delta}{\epsilon^i}]$ for some $\delta \in \{1, 2^1, 2^2, \dots, 2^{\lceil \log(1/\epsilon) \rceil}\}$, there exists a

(δ, ε)-PIT $T \in \mathcal{T}$ such that a node (x, y, i) at level i of T is $O(\varepsilon)$ -close to pair (x_0, y_0) . Recall that (x, y, i) is $O(\varepsilon)$ -close to (x_0, y_0) means $x_0, y_0 \in C_i(x, y)$, and for any point $t \in C_i(x, y)$, $d_X(t, x)$ or $d_X(t, y)$ are at most $O(\varepsilon)d_X(x_0, y_0)$.

Also recall from Definition 3 that T satisfies the covering property:

- [Children covering.] If (u, v, j) is a child of (x, y, i) for $j < i$, then $d_X(u, \{x, y\})$ and $d_X(v, \{x, y\})$ are $O(\frac{\delta}{\varepsilon^{i-1}})$.
- [Bounded diameter.] The cluster $C_i(x, y)$ of (x, y, i) is the union of all leaf labels in the subtree rooted as (x, y, i) , and furthermore, the diameter of $C_i(x, y)$ is $O(\frac{\delta}{\varepsilon^i})$.

Lemma 23 below concludes the stretch analysis, implying Theorem 5.

Lemma 23. For every pair (x_0, y_0) whose distance $d_X(x_0, y_0) \in [\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i}]$, there exists a tree $J \in \mathcal{J}$ containing a path π from the leaf of x_0 to the leaf of y_0 such that the total weight of the edges along π is at most $(1 + O(\varepsilon))d_X(x_0, y_0)$.

Proof: For any pair $x_0, y_0 \in S$ whose $d_X(x_0, y_0) \in [\frac{\delta}{\varepsilon^i}, \frac{2\delta}{\varepsilon^i}]$, by the pairwise covering property of \mathcal{T} , there exists a tree $J \in \mathcal{J}$ that covers (x_0, y_0) . That is, J has a node $a = (x, y, i)$ where (x, y, i) is $O(\varepsilon)$ -close to (x_0, y_0) .

Suppose that $d_X(x_0, \{x, y\})$ and $d_X(y_0, \{x, y\})$ are at most $c\varepsilon d_X(x_0, y_0)$ for some constant c . First, we argue that $d_X(x, y) = \Theta(\frac{\delta}{\varepsilon^i})$. By triangle inequality, we obtain:

$$\begin{aligned} d_X(x_0, y_0) - 2c\varepsilon d_X(x_0, y_0) &\leq d_X(x, y) \leq d_X(x_0, y_0) + 2c\varepsilon d_X(x_0, y_0) \\ \Leftrightarrow (1 - 2c\varepsilon)\frac{\delta}{\varepsilon^i} &\leq d_X(x, y) \leq (2 + 2c\varepsilon)\frac{\delta}{\varepsilon^i} \end{aligned} \tag{10}$$

The path π from x_0 to y_0 in J travels from the leaf of x_0 to (x, y, i) , then from (x, y, i) down to the leaf of y_0 . Let $l_J(x_0), l_J(y_0)$ be the leaf of x_0, y_0 in J . Since $d_X(x, y) \leq (1 + O(\varepsilon))d_X(x_0, y_0)$, if $w(l_J(x_0), l_J(y_0)) \leq O(\varepsilon)d_X(x, y) + d_X(x, y)$, the stretch follows.

To compute $w(l_J(x_0), l_J(y_0))$, we bound the total weight of edges in J from a leaf to a child of $a = (x, y, i)$. Consider a child $b = (u, v, j)$ of $a = (x, y, i)$ where $j < i$, and a point $t \in C_j(u, v)$.

Claim 8. If b is the ancestor at level j of $l_J(t)$, then $w(l_J(t), b) = O(\frac{\delta}{\varepsilon^j})$

Proof: Consider the base case where b is the parent of $l_J(t)$. By the construction of J , we have two cases of $w(l_J(t), b)$ when $d_X(u, t) \leq d_X(v, t)$:

- If $d_X(u, \{x, y\}) \leq d_X(v, \{x, y\})$, then $w(l_J(t), b) = d_X(t, u)$.
- If $d_X(u, \{x, y\}) > d_X(v, \{x, y\})$, then $w(l_J(t), b) = d_X(t, u) + d_X(u, v)$.

By the children covering of J , $d_X(t, u) = O(\frac{\delta}{\varepsilon^{j-1}})$, and by the bounded diameter property, $d_X(t, v) = O(\frac{\delta}{\varepsilon^j})$ and $d_X(u, v) = O(\frac{\delta}{\varepsilon^j})$. Thus $w(l_J(t), b) = O(\frac{\delta}{\varepsilon^j})$.

Similarly, if b is the parent of $l_J(t)$ and $d_X(u, t) > d_X(v, t)$, by the construction of J , we have two cases:

- If $d_X(u, \{x, y\}) \leq d_X(v, \{x, y\})$, then $w(l_J(t), b) = d_X(t, v) + d_X(u, v)$.
- If $d_X(u, \{x, y\}) > d_X(v, \{x, y\})$, then $w(l_J(t), b) = d_X(t, v)$.

2291 By the children covering and the bounded diameter properties, we again obtain $w(l_J(t), b) = O(\frac{\delta}{\varepsilon^j})$.
 2292

2293 Now, for the inductive case where b is an ancestor at level j of $l_J(t)$. By induction, let $b' = (u', v', j')$
 2294 be a child of b that is the ancestor at level $j' < j$ of $l_J(t)$, and that $w(l_J(t), b') = O(\frac{\delta}{\varepsilon^{j'}})$. By the
 2295 weight update in Step 3, $w(b, b') \leq d_X(\{u, v\}, \{u', v'\}) + d_X(u, v)$. By the children covering property of J ,
 $d_X(\{u, v\}, \{u', v'\}) = O(\frac{\delta}{\varepsilon^{j-1}})$, and by the bounded diameter property, $d_X(u, v) = O(\frac{\delta}{\varepsilon^j})$. We obtain:

$$\begin{aligned} w(l_J(t), b) &= w(l_J(t), b') + w(b', b) \\ &\leq O(\frac{\delta}{\varepsilon^{j'}}) + d_X(\{u, v\}, \{u', v'\}) + d_X(u, v) \\ &= O(\frac{\delta}{\varepsilon^{j-1}}) + O(\frac{\delta}{\varepsilon^{j-1}}) + O(\frac{\delta}{\varepsilon^j}) \\ &= O(\frac{\delta}{\varepsilon^j}) \end{aligned} \tag{11}$$

2296 as desired. \square

2297 Back to proving the stretch, observe that x_0 (or y_0) can not be within the distance $O(\frac{\delta}{\varepsilon^{i-1}})$ to both x and
 2298 y , since $d_X(x_0, y_0)$ and $d_X(x, y)$ are $\Theta(\frac{\delta}{\varepsilon^i})$. Without loss of generality, suppose that $d_X(x, x_0) = O(\frac{\delta}{\varepsilon^{i-1}})$
 2300 and $d_X(y, y_0) = O(\frac{\delta}{\varepsilon^{i-1}})$. Let $b_x = (u_x, v_x, j_x)$ be a child of $a = (x, y, i)$ and the ancestor of $l_J(x_0)$. By
 2301 the bounded diameter property, $d_X(x_0, u_x)$ and $d_X(x_0, v_x)$ are $O(\frac{\delta}{\varepsilon^{j_x}}) = O(\frac{\delta}{\varepsilon^{i-1}})$ since $j_x < i$. By triangle
 2302 inequality, $d_X(u_x, x)$ and $d_X(v_x, x)$ are at most $O(\frac{\delta}{\varepsilon^{i-1}})$. Similarly, let $b_y = (u_y, v_y, j_y)$ be a child of
 2303 $a = (x, y, i)$ and the ancestor of $l_J(y_0)$, we have $d_X(u_y, y)$ and $d_X(v_y, y)$ are at most $O(\frac{\delta}{\varepsilon^{i-1}})$. We obtain
 2304 that labeled points of b_x are close to x , and labeled points of b_y are close to y . Thus, there is only one
 2305 node between b_x and b_y such that we add $d_X(x, y)$ to $w(b_x, a)$ or $w(b_y, a)$ in Step 3. We obtain:

$$w(b_x, a) + w(a, b_y) = d_X(\{u_x, v_x\}, \{x, y\}) + d_X(\{x, y\}, \{u_y, v_y\}) + d_X(x, y) \tag{12}$$

2306 By the children covering property of J , $d_X(\{x, y\}, \{u_x, v_x\}) = O(\frac{\delta}{\varepsilon^{i-1}})$ and $d_X(\{x, y\}, \{u_y, v_y\}) = O(\frac{\delta}{\varepsilon^{i-1}})$. Thus, $w(b_x, a) + w(a, b_y) = O(\frac{\delta}{\varepsilon^{i-1}}) + d_X(x, y)$. Besides that, by Claim 8, $w(l_J(x_0), b_x) = O(\frac{\delta}{\varepsilon^{i-1}})$
 2307 and $w(l_J(y_0), b_y) = O(\frac{\delta}{\varepsilon^{i-1}})$, since $j_x, j_y \leq i-1$. The total weight of edges in π is bounded as follows:

$$\begin{aligned} w(l_J(x_0), l_J(y_0)) &= w(l_J(x_0), b_x) + (w(b_x, a) + w(a, b_y)) + w(b_y, l_J(y_0)) \\ &= O(\frac{\delta}{\varepsilon^{i-1}}) + \left(O(\frac{\delta}{\varepsilon^{i-1}}) + d_X(x, y) \right) + O(\frac{\delta}{\varepsilon^{i-1}}) \\ &\leq (1 + O(\varepsilon))d_X(x_0, y_0) \end{aligned} \tag{13}$$

2312 where the last inequality holds since $d_X(x, y) \leq (1 + O(\varepsilon))d_X(x_0, y_0)$ by Equation (10) and $d_X(x_0, y_0) =$
 2313 $\Theta(\frac{\delta}{\varepsilon^i})$.

2314 7.3 Dynamic Closest Pair

2315 **Theorem 14.** *Given a dynamic point set S in a doubling metric of dimension λ , there is a data structure
 2316 that maintains the closest pair for S in $O_\lambda(\log(n))$ time per update.*

2317 **Proof:** The data structure consists of a $(\varepsilon^{-O(\lambda)}, \varepsilon)$ -LSO Σ for $\varepsilon = 1/2$, and a min-heap H , where H
 2318 maintains all pairs (u, v) such that u and v are adjacent in an ordering of Σ , and is keyed by the distances
 2319 between the pair. The closest pair is determined by the pair with the minimum distance in H .

2320 If q is inserted into S , we invoke $\text{INSERT}(q, \Sigma)$, then for every ordering $\sigma_i \in \Sigma$, we find the successor
 2321 s_i and the predecessor p_i of q in σ_i by calling $\text{GETSUCCESSOR}(q, i, \Sigma)$ and $\text{GETPREDECESSOR}(q, i, \Sigma)$.
 2322 Next, we insert two pairs (p_i, q) and (q, s_i) to H , and remove the pair (p_i, s_i) from H .

If q is deleted from S , for every ordering $\sigma_i \in \Sigma$, we find the successor s_i and the predecessor p_i of q in σ_i . We remove (p_i, q) and (q, s_i) from H , and add (p_i, s_i) to H .

First, we analyze the running time. By Theorem 1, the $(\varepsilon^{-O(\lambda)}, \varepsilon)$ -LSO Σ runs in $O(\varepsilon^{-O(\lambda)} \log n)$ per update. Getting the successor and the predecessor of a point in an ordering takes $O(1)$, in all orderings takes $O(|\Sigma|) = \varepsilon^{-O(\lambda)}$. When we insert or delete a point in Σ , there are $O(|\Sigma|) = \varepsilon^{-O(\lambda)}$ pairs updated (inserted or deleted) in H . H maintains $O(n|\Sigma|)$ pairs, thus its running time is $O(\log |\Sigma| + \log n)$ per insertion or deletion of a pair. Since $\varepsilon = 1/2$, the total running time per update of the data structure maintaining the closest pair for S is $2^{O(\lambda)}(\log n)$, as claimed.

Next, we show the correctness. It suffices to get the predecessor and the successor of q only since Σ is stable by Theorem 1. Observe that if (a, b) is the closest pair, then by the definition of LSO, there exists an ordering in Σ where a and b are adjacent, which means (a, b) is maintained in H . Suppose otherwise, there exists an ordering $\sigma \in \Sigma$ and a point u where $a \prec_\sigma u \prec_\sigma b$ where $\min\{d_X(a, u), d_X(b, u)\} \leq \varepsilon \cdot d_X(a, b) < d_X(a, b)$ as $\varepsilon = 1/2$. Then either u is closer to a than b or u is closer to b than a ; both cases contradict the fact that (a, b) is the closest pair. \square

7.4 Approximate Bichromatic Closest Pair

Theorem 7. *Given a parameter $\varepsilon \in (0, 1)$ and two dynamic point sets R, B in doubling metric of dimension λ , there is a data structure such that it maintains $(1 + \varepsilon)$ -closest pair (r, b) where $r \in R, b \in B$, and runs in $O(\varepsilon^{-O(\lambda)} \log(n))$ per update of R or B , where $n = |R| + |B|$.*

Proof: We can find $(1 + \varepsilon)$ -approximation for the bichromatic closest pair under insertions and deletions of B and R by using a min-heap H , and a $(\varepsilon^{-O(\lambda)}, \varepsilon)$ -LSO Σ . The key idea is we apply the LSO to find adjacent pairs (r, b) where $r \in R, b \in B$ in every ordering $\sigma \in \Sigma$, and then use H to maintain these pairs sorted by descending order of $d_X(r, b)$.

To find the adjacent pair from a new point, suppose that r is newly added to R . First, we invoke $\text{INSERT}(r, \Sigma)$. After that, we invoke $\text{GETSUCCESSOR}(r, i, \Sigma)$ to find the successor s of r , and $\text{GETPREDECESSOR}(r, i, \Sigma)$ to find the predecessor p of r in σ . Now we obtain $p \prec_\sigma r \prec_\sigma s$. If $s \in B$, we add (r, s) to H . If $p \in R$ and $s \in B$, we remove the pair (p, s) from H . For a new point $b \in B$, we invoke $\text{INSERT}(b, \Sigma)$, then we follow a similar way to find $p \prec_\sigma b \prec_\sigma s$. If $p \in R$, we update (p, b) to H . If $p \in R$ and $s \in B$, we remove (p, s) from H .

With deletion, when a point $r \in R$ is deleted, we retrieve from r the predecessor p and the successor s in σ to obtain $p \prec_\sigma r \prec_\sigma s$. If $s \in B$, then we remove (r, s) out of H . If $p \in R$ and $s \in B$, we add (p, s) to H . After that, we invoke $\text{DELETE}(r, \Sigma)$ to remove r out of all orderings in Σ . Similarly when a point $b \in B$ is deleted, we retrieve its predecessor p and successor s in σ : $p \prec_\sigma b \prec_\sigma s$. If $p \in R$, we remove (p, b) out of H . If $p \in R$ and $s \in B$, we add back to H the pair (p, s) . Finally, we invoke $\text{DELETE}(b, \Sigma)$.

Since each point is adjacent to at most 2 other points with different colors, and we have $\tau = \varepsilon^{-O(\lambda)}$ orderings, H maintains at most $\varepsilon^{-O(\lambda)}O(n)$ pairs. Thus, operations of insertion and deletion in H run in $O(\lambda \log \frac{1}{\varepsilon} + \log n)$ time per update. By Theorem 1, $(\varepsilon^{-O(\lambda)}, \varepsilon)$ -LSO Σ runs in $O(\varepsilon^{-O(\lambda)} \log n)$ time per update and $O(1)$ per predecessor/successor query. Therefore, the running time totally is $O(\varepsilon^{-O(\lambda)} \log n)$ per update as claimed.

To prove the correctness, we consider the closest pair (r, b) . By Theorem 1, Σ is stable, thus it suffices to query the predecessor and the successor of a new point or deleted point only. By the definition of LSO, there is an ordering σ such that: for every point t where $r \prec_\sigma t \prec_\sigma b$, $d_X(r, t) \leq \varepsilon d_X(r, b)$ or $d_X(t, b) \leq \varepsilon d_X(r, b)$. Observe that if $d_X(r, t) \leq \varepsilon d_X(r, b)$, t must be a point in R , otherwise (r, b) is not the closest bichromatic pair. Similarly, if $d_X(t, b) \leq \varepsilon d_X(r, b)$, t must be a point in B . Thus in σ from r to b , there is an adjacent pair (r', b') such that $r' \in R, b' \in B$, and both $d_X(r, r')$, $d_X(b', b)$ are at most $\varepsilon d_X(r, b)$. By triangle inequality, we have:

2368

$$\begin{aligned}
 d_X(r', b') &\leq d_X(r, b) + d_X(r, r') + d_X(b', b) \\
 &\leq d_X(r, b) + \varepsilon d_X(r, b) + \varepsilon d_X(r, b) \\
 &= (1 + 2\varepsilon)d_X(r, b).
 \end{aligned} \tag{14}$$

2369 Since (r', b') is maintained in H , we correctly find a pair with distance at most $(1 + 2\varepsilon)d_X(r, b)$. Adjusting
2370 ε by a constant factor, the theorem follows. \square

2371 7.5 Dynamic Approximate Nearest Neighbors

2372 **Theorem 8.** *Given a dynamic point set S in doubling metrics of dimension λ , we can construct a $(1 + \varepsilon)$ -
2373 nearest neighbor data structure for supporting point deletions/insertions in $O(\varepsilon^{-O(\lambda)} \log(n))$ time per
2374 update, and $\varepsilon^{-O(\lambda)} \log(n)$ query time.*

2375 **Proof:** We can directly use an $(\varepsilon^{-O(\lambda)}, \varepsilon)$ -LSO Σ to find approximate nearest neighbours. When we add
2376 a new point q to S , we invoke $\text{INSERT}(q, \Sigma)$; when we delete an existing point q , we invoke $\text{DELETE}(q, \Sigma)$.
2377 To find an approximate nearest neighbour of x , we follow 3 steps: (1) insert x to Σ by $\text{INSERT}(x, \Sigma)$, (2)
2378 for each ordering $\sigma_i \in \Sigma$, find the predecessor p_i and the successor s_i of x , then return the point who is
2379 closest to x among $\{p_1, s_1, \dots, p_{|\Sigma|}, s_{|\Sigma|}\}$, (3) remove x out of Σ by $\text{DELETE}(x, \Sigma)$. Note that if x is a point
2380 that we already add into S , we run only step (2).

2381 By Theorem 1, Σ has $O(\varepsilon^{-O(\lambda)} \log(n))$ time per update, and we invoke $2\varepsilon^{-O(\lambda)}$ predecessor and
2382 successor queries, each takes $O(1)$ time. Thus we obtain the running time per update and per query as
2383 claimed.

2384 Now to show the correctness. Given a query point x , consider the closest point y of x . By the
2385 definition of LSO, there is an ordering σ such that any point t between x and y has $d_X(t, x) \leq \varepsilon d_X(x, y)$
2386 or $d_X(t, y) \leq \varepsilon d_X(x, y)$. Without loss of generality, suppose that $x \prec_\sigma y$. Since y is the point closest
2387 to x , thus (i) x and y must be adjacent in σ , (ii) or any point t between x and y in σ must have
2388 $d_X(t, x) \geq d_X(x, y)$ and $d_X(t, y) \leq \varepsilon d_X(x, y)$. For (i), we are done. For (ii), consider the successor s of
2389 x in σ . By triangle inequality, we obtain:

2390

$$\begin{aligned}
 d_X(x, s) &\leq d_X(x, y) + d_X(y, s) \\
 &\leq d_X(x, y) + \varepsilon d_X(x, y) \\
 &= (1 + \varepsilon)d_X(x, y)
 \end{aligned} \tag{15}$$

2391 Similarly, when $y \prec_\sigma x$, we consider the predecessor p of x in σ and obtain:

2392

$$d_X(x, p) \leq (1 + \varepsilon)d_X(x, y) \tag{16}$$

2393 We return a point x' adjacent with x in an ordering of Σ such that $d_X(x, x') \leq d_X(x, s)$ and $d_X(x, x') \leq$
2394 $d_X(x, p)$. Thus x' is $(1 + \varepsilon)$ -approximate nearest neighbour of x . \square

2395 **Acknowledgements.** This work was supported by the NSF CAREER Award No. CCF-223728, an NSF
2396 Grant No. CCF-2121952, and a Google Research Scholar Award. Thank Aditya Kumar Roy Chowdhury
2397 for joining the early states of this work.

2398 References

2399 [1] Sunil Arya, Gautam Das, David M. Mount, Jeffrey S. Salowe, and Michiel Smid. Euclidean spanners.
2400 In *Proceedings of the 27th Annual ACM symposium on Theory of Computing*, STOC '95, 1995.
2401 [doi:10.1145/225058.225191](https://doi.org/10.1145/225058.225191).

2402 [2] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal
 2403 algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*,
 2404 45(6):891–923, 1998. [doi:10.1145/293347.293348](https://doi.org/10.1145/293347.293348).

2405 [3] Yair Bartal, Ora Nova Fandina, and Ofer Neiman. Covering metric spaces by few trees. *Journal of*
 2406 *Computer and System Sciences*, 130:26–42, 2022. [doi:10.1016/j.jcss.2022.06.001](https://doi.org/10.1016/j.jcss.2022.06.001).

2407 [4] S. N. Bespamyatnikh. An optimal algorithm for closest-pair maintenance. *Discrete & Computational*
 2408 *Geometry*, 19(2):175–195, 1998. [doi:10.1007/pl00009340](https://doi.org/10.1007/pl00009340).

2409 [5] Kevin Buchin, Sariel Har-Peled, and Dániel Oláh. A Spanner for the Day After. In *35th International*
 2410 *Symposium on Computational Geometry (SoCG 2019)*, pages 19:1–19:15, 2019. [doi:10.4230/LIPIcs.SoCG.2019.19](https://doi.org/10.4230/LIPIcs.SoCG.2019.19).

2412 [6] Kevin Buchin, Sariel Har-Peled, and Dániel Oláh. Sometimes Reliable Spanners of Almost Linear
 2413 Size. In *28th Annual European Symposium on Algorithms (ESA 2020)*, pages 27:1–27:15, 2020.
 2414 [doi:10.4230/LIPIcs.ESA.2020.27](https://doi.org/10.4230/LIPIcs.ESA.2020.27).

2415 [7] T.-H. Hubert Chan, Anupam Gupta, Bruce M. Maggs, and Shuheng Zhou. On hierarchical routing
 2416 in doubling metrics. *ACM Transactions on Algorithms*, 12(4):1–22, 2016. [doi:10.1145/2915183](https://doi.org/10.1145/2915183).

2417 [8] T.-H. Hubert Chan, Mingfei Li, and Li Ning. Sparse fault-tolerant spanners for doubling metrics
 2418 with bounded hop-diameter or degree. In *ICALP (1)*, pages 182–193, 2012. Full version in
 2419 *Algorithmica*’15.

2420 [9] T.-H. Hubert Chan, Mingfei Li, Li Ning, and Shay Solomon. New doubling spanners: Better and
 2421 simpler. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Proc.*
 2422 *40th ICALP*, pages 315–327, 2013. Full version in *SICOMP*’15. [doi:10.1007/978-3-642-39206-1_27](https://doi.org/10.1007/978-3-642-39206-1_27).

2424 [10] Timothy M. Chan, Sariel Har-Peled, and Mitchell Jones. On locality-sensitive orderings and their
 2425 applications. *SIAM Journal on Computing*, 49(3):583–600, 2020. [doi:10.1137/19M1246493](https://doi.org/10.1137/19M1246493).

2426 [11] Hsien-Chih Chang, Jonathan Conroy, Hung Le, Lazar Milenkovic, Shay Solomon, and Cuong Than.
 2427 Covering planar metrics (and beyond): $O(1)$ trees suffice, 2023. [doi:10.48550/ARXIV.2306.06215](https://doi.org/10.48550/ARXIV.2306.06215).

2429 [12] Richard Cole and Lee-Ad Gottlieb. Searching dynamic point sets in spaces with bounded doubling
 2430 dimension. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*,
 2431 STOC ’06, page 574–583, New York, NY, USA, 2006. Association for Computing Machinery. [doi:10.1145/1132516.1132599](https://doi.org/10.1145/1132516.1132599).

2433 [13] Artur Czumaj and Hairong Zhao. Fault-tolerant geometric spanners. *Discrete & Computational*
 2434 *Geometry*, 32(2), 2004. [doi:10.1007/s00454-004-1121-7](https://doi.org/10.1007/s00454-004-1121-7).

2435 [14] D. Eppstein. Dynamic euclidean minimum spanning trees and extrema of binary functions. *Discrete*
 2436 *& Computational Geometry*, 13(1):111–122, 1995. [doi:10.1007/bf02574030](https://doi.org/10.1007/bf02574030).

2437 [15] Arnold Filtser. Labeled Nearest Neighbor Search and Metric Spanners via Locality Sensitive
 2438 Orderings. In *39th International Symposium on Computational Geometry (SoCG 2023)*, pages
 2439 33:1–33:18, 2023. [doi:10.4230/LIPIcs.SoCG.2023.33](https://doi.org/10.4230/LIPIcs.SoCG.2023.33).

[16] Arnold Filtser and Hung Le. Locality-sensitive orderings and applications to reliable spanners. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, page 1066–1079, New York, NY, USA, 2022. Association for Computing Machinery. [doi:10.1145/3519935.3520042](https://doi.org/10.1145/3519935.3520042).

[17] Mordecai Golin, Rajeev Raman, Christian Schwarz, and Michiel Smid. Randomized data structures for the dynamic closest-pair problem. *SIAM Journal on Computing*, 27(4):1036–1072, 1998. [doi:10.1137/s0097539794277718](https://doi.org/10.1137/s0097539794277718).

[18] Lee-Ad Gottlieb and Liam Roditty. Improved algorithms for fully dynamic geometric spanners and geometric routing. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, page 591–600. Society for Industrial and Applied Mathematics, 2008.

[19] Lee-Ad Gottlieb and Liam Roditty. An optimal dynamic spanner for doubling metric spaces. In *Proceedings of the 16th European Symposium on Algorithms*, ESA '08, pages 478–489, 2008.

[20] Sariel Har-Peled and Manor Mendel. Fast construction of nets in low-dimensional metrics and their applications. *SIAM J. Comput.*, 35(5):1148–1184, 2006.

[21] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999. [doi:10.1145/320211.320215](https://doi.org/10.1145/320211.320215).

[22] Dorit S. Hochbaum and Wolfgang Maass. Approximation schemes for covering and packing problems in image processing and vlsi. *Journal of the ACM*, 32(1):130–136, 1985. [doi:10.1145/2455.214106](https://doi.org/10.1145/2455.214106).

[23] Camille Jordan. Sur les assemblages de lignes. 1869.

[24] Omri Kahalon, Hung Le, Lazar Milenković, and Shay Solomon. Can't see the forest for the trees: Navigating metric spaces by bounded hop-diameter spanners. In *Proceedings of the 41st ACM Symposium on Principles of Distributed Computing*, PODC '22. ACM, 2022. [doi:10.1145/3519270.3538414](https://doi.org/10.1145/3519270.3538414).

[25] Tsvi Kopelowitz and Moshe Lewenstein. Dynamic weighted ancestors. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, page 565–574. Society for Industrial and Applied Mathematics, 2007.

[26] Robert Krauthgamer and James R. Lee. Navigating nets: Simple algorithms for proximity search. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, page 798–807, 2004.

[27] Hung Le, Shay Solomon, and Cuong Than. Optimal fault-tolerant spanners in euclidean and doubling metrics: Breaking the $\omega(\log n)$ lightness barrier. In *Proceedings of the 64th IEEE Symposium on the Foundations of Computer Science*, 2023. To appear in FOCS'23.

[28] Christos Levcopoulos, Giri Narasimhan, and Michiel Smid. Efficient algorithms for constructing fault-tolerant geometric spanners. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, STOC'98, pages 186–195, 1998.

[29] T. Lukovszki. New results on fault tolerant geometric spanners. In *The 6th International Workshop on Algorithms and Data Structures*, WADS '99, pages 193–204, 1999. [doi:10.1007/3-540-48447-7_20](https://doi.org/10.1007/3-540-48447-7_20).

2480 [30] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *Proceedings of*
2481 *the third annual ACM-SIAM symposium on Discrete algorithms*, pages 367–375, 1992.

2482 [31] Giri Narasimhan and Michiel Smid. *Geometric Spanner Networks*. Cambridge University Press,
2483 2007. [doi:10.1017/cbo9780511546884](https://doi.org/10.1017/cbo9780511546884).

2484 [32] L. Roditty. Fully dynamic geometric spanners. *Algorithmica*, 62(3-4):1073–1087, 2011. [doi:10.1007/s00453-011-9504-7](https://doi.org/10.1007/s00453-011-9504-7).

2485 [33] Michiel Smid, Anil Maheshwari, and Wolfgang Mulzer. A simple randomized $o(n \log n)$ -time
2486 closest-pair algorithm in doubling metrics. *Journal of Computational Geometry*, page Vol. 11 No. 1
2487 (2020), 2020. [doi:10.20382/JOCG.V11I1A20](https://doi.org/10.20382/JOCG.V11I1A20).

2488 [34] Shay Solomon. From hierarchical partitions to hierarchical covers: optimal fault-tolerant spanners
2489 for doubling metrics. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*,
2490 STOC’14, pages 363–372, 2014.

2491 [35] R.E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in
2492 logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer*
2493 *Science*., FOCS ’84. IEEE, 1984. [doi:10.1109/sfcs.1984.715896](https://doi.org/10.1109/sfcs.1984.715896).

2494 [36] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24,
2495 2005. [doi:10.1145/1044731.1044732](https://doi.org/10.1145/1044731.1044732).

2496