Hierarchical Structures for High Dimensional Data Analysis

Mahmoodreza Jahanseirroodsari

University of Connecticut, mahmoodreza.jahanseirroodsari@uconn.edu

Follow this and additional works at: https://opencommons.uconn.edu/dissertations

Recommended Citation
Hierarchical Structures for High Dimensional Data Analysis

Mahmoodreza Jahanseirroodsari, Ph.D.
University of Connecticut, 2019

The volume of data is not the only problem in modern data analysis, data complexity is often more challenging. In many areas such as computational biology, topological data analysis, and machine learning, the data resides in high dimensional spaces which may not even be Euclidean. Therefore, processing such massive and complex data and extracting some useful information is a big challenge. Our methods will apply to any data sets given as a set of objects and a metric that measures the distance between them.

In this dissertation, we first consider the problem of preprocessing and organizing such complex data into a hierarchical data structure that allows efficient nearest neighbor and range queries. There have been many data structures for general metric spaces, but almost all of them have construction time that can be quadratic in terms of the number of points. There are only two data structures with $O(n \log n)$ construction time, but both have very complex algorithms and analyses. Also, they cannot be implemented efficiently. Here, we present a simple, randomized incremental algorithm that builds a metric data structure in $O(n \log n)$ time in expectation. Thus, we achieve the best of both worlds, simple implementation with asymptotically optimal performance.

Furthermore, we consider the close relationship between our metric data structure and point orderings used in applications such as $k$-center clustering. We give linear time algorithms to go back and forth between these orderings and our metric data
In the last part, we use metric data structures to extract topological features of a data set, such as the number of connected components, holes, and voids. We give an efficient algorithm for constructing a \((1 + \varepsilon)\)-approximation to the so-called Nerve filtration of a metric space, a fundamental tool in topological data analysis.
Hierarchical Structures for High Dimensional Data Analysis

Mahmoodreza Jahanseirroodsari

B.S., Islamic Azad University, 2009
M.S., Amirkabir University of Technology, 2012

A Dissertation
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy
at the
University of Connecticut

2019
APPROVAL PAGE

Doctor of Philosophy Dissertation

Hierarchical Structures for High Dimensional Data Analysis

Presented by

Mahmoodreza Jahanseirroodsari, B.S., M.S.

Major Advisor

______________________________

Donald R. Sheehy

Associate Advisor

______________________________

Thomas J. Peters

Associate Advisor

______________________________

Sanguthevar Rajasekaran

University of Connecticut

2019
ACKNOWLEDGEMENTS

My deepest gratitude goes to my major advisor, Professor Donald Sheehy, for his excellent guidance, encouragement, and continuous support during my Ph.D. study. His immense knowledge, motivation, and patience helped me in all the time of my research endeavors and writing of this thesis. Despite Don’s heavy schedule, his door was always open to me whenever I ran into a trouble spot and was in need of discussion. I could not have imagined having a better advisor and mentor for my Ph.D study.

Additionally, I would like to thank my thesis committee Professor Thomas Peters and Professor Sanguthevar Rajasekaran for their time, advice, and insightful comments. My thanks also go to my colleagues, Nicholas Cavanna and Kirk Gardner for their discussion and comments in my research. I would also like to extend my thanks to the CSE Department including Rebecca Randazzo, Joy Billion, Lara Chiaverini, and Howard Ellis for their help from my very first day in the program.

Finally, I want to thank my family and friends for their encouragement and support. To the memory of my parents, whom I learned to be diligent, devoted, and strong. You are gone but your belief in me has made this journey possible. My special thanks go to my beloved wife, Mahtab, without whom none of this would be possible. With her endless love and support, I have the motivation to complete the doctoral program.
3.3.1 Coarsening .................................................. 31
3.3.2 Refining ...................................................... 35

4 Net-Tree Construction ........................................... 39

4.1 Approximate Voronoi Diagrams from Net-Trees ................. 40
4.2 Bottom-up Insertion into a Net-Tree .......................... 42
  4.2.1 Insertion ..................................................... 43
  4.2.2 The Bottom-up Propagation Algorithm ................. 48
  4.2.3 Analysis of the Construction ......................... 51
  4.2.4 Point Location .......................................... 52
4.3 Randomized Incremental Construction ............................ 58
  4.3.1 The Point Location Algorithm ....................... 59
  4.3.2 Analysis of the Point Location Algorithm ........ 63
4.4 Experimental Results ........................................... 80
  4.4.1 Model Comparison Metrics .............................. 81
  4.4.2 Discussion ................................................ 88

5 Net-Trees and Permutations ...................................... 99

5.1 Greedy Permutations and Their Relatives ...................... 100
5.2 Transition between Greedy Permutations and Net-Trees ...... 104
  5.2.1 Net-Trees from Locally Greedy Permutations ....... 104
  5.2.2 Locally Greedy Permutations from Net-Trees ....... 106

6 Hierarchical Metric Trees for Topological Data Analysis .......... 108

6.1 Perturbed Distances ........................................... 110
6.2 Sparse Filtrations ............................................ 114
6.3 Algorithms to Construct Sparse Filtrations .................. 116
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.1</td>
<td>Finding Neighborhoods Suffices</td>
<td>117</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Constructing the Neighborhood Graph</td>
<td>118</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Higher Dimensional Simplices</td>
<td>124</td>
</tr>
<tr>
<td>6.4</td>
<td>Removing Vertices</td>
<td>126</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion and Future Work</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>131</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Hierarchical trees are powerful data structures to solve classical computational geometry problems such as approximate nearest neighbor and range search. For Euclidean spaces, quadtrees [32] and k-d trees [8] are perhaps the two most famous data structures. See [2, 4, 5, 17, 38] for the progress in answering proximity searches in Euclidean spaces. However, quadtrees and k-d trees are not directly applicable for general metric spaces because their construction is based on partitioning of a spaces into several axis aligned rectangles. Most data structures for general metric spaces are generalizations of their Euclidean counterparts. In fact, they leverage the hierarchical partitioning, but they use metric balls for the partitioning step.

In any metric spaces, the construction times or sizes of data structures depend on the dimension. Many of data structures suffer from the curse of dimensionality, that is the time or space complexity increases exponentially with the dimension. There is a line of research to eliminate the exponential dependence on the dimension primarily using locality sensitive hashing and dimensionality reduction techniques [1, 40, 45, 54, 59]. In this dissertation, we assume that the input has a low intrinsic dimension although its ambient dimension can be big. As an example, a set of points on a line in a $d$-
dimensional Euclidean space has the ambient dimension $d$, but its intrinsic dimension is 1. An intrinsic dimension for metric spaces will be introduced in Chapter 2.

Har-Peled and Mendel introduced the net-tree as a linear-size data structure that efficiently solves a variety of metric problems such as approximate nearest neighbor search, well-separated pair decomposition, spanner construction, and others [41]. Net-trees are similar to several other data structures that store points in hierarchies of metric nets (subsets satisfying some packing and covering constraints) arranged into a tree or DAG. Examples include navigating nets [52], cover trees [9], dynamic hierarchical spanners [24, 36], and deformable spanners [33].

In Chapter 3, first we introduce two variations of net-trees which greatly simplify our algorithms and analyses. We substitute the global conditions of net-trees with local conditions and show that local net-trees also satisfy global properties of net-trees. Then, we present a linear time algorithm that converts a cover tree [9] into a net-tree, where a cover tree is the simplest metric data structure that works efficiently in practice and has been used widely for machine learning applications [25, 26, 46, 55, 60, 65, 68] (see Section 2.3). This conversion results in better theoretical guarantees for cover trees, but it comes at the cost of larger parameters and increased size. We also propose two algorithms to modify a net-tree that make a trade-off between the height and the node degree.

There are two known algorithms for building a net-tree [41] or a closely related structure [36] in $O(n \log n)$ expected time for doubling metrics (i.e. metrics with a low intrinsic dimension, see Section 2.1). Both are quite complex and are primarily of theoretical interest. A much simpler algorithm due to Clarkson [21] can be combined with an algorithm of Har-Peled and Mendel [41] to run in $O(n \log \Delta)$ time, where $\Delta$ is the spread of the input, i.e. the ratio of the largest to smallest pairwise distances. Most of the complications of the theoretical algorithm are to eliminate this dependence on
the spread.

In Chapter 4, we combine the conceptual simplicity of Clarkson’s idea with a simple randomized incremental algorithm to achieve the same $O(n \log n)$ running time of the best theoretical algorithms. The main improvement over the related data structures [24, 41] that can be computed in $O(n \log n)$ time is the increased simplicity. In particular, the point location structure, the primary bottleneck in all related work, is just a dictionary mapping uninserted points to nodes in the tree and is very easy to update. As testament to the simplicity, a readable implementation in python has only 300 lines of code and is available online [47]. In Section 4.4, we show how the code works on several point sets designed to reveal the geometric properties that govern the true performance in practice. We also found that our randomized incremental point location scheme was faster than approached based on prior work. The experiments show that point sets exhibiting large changes of scale (and thus producing deep trees) are constructed much faster with our new algorithm. These experiments also confirm the expectation that constant factors depending exponentially on the dimension can be significant even in 3-dimensions.

Another improvement is in the tighter bounds on the so-called relative constant. This is the constant factor that bounds the ratio of distances between relative links, the edges stored between nearby nodes in the same level of the tree. These relative links form a hierarchical spanner, and their number dominates the space complexity of the data structure. In similar constructions used in topological data analysis, it was found that although a relative constant of 10 was needed for a particular algorithm, the space blowup required using a constant closer to 3 in practice, sacrificing theoretical guarantees [58]. In previous work the relative constant was 13 or more. In this dissertation, we show that the relative constant can be pushed towards 2 as a function of the difference in scales between adjacent levels in the tree. A value of 6 is easily
achievable in practice and seems to work well. The experiments show that this is a major factor in both the construction time and size of net-trees.

The constructions of hierarchical trees are closely related to greedy permutations, also known as farthest point sampling, and approximate k-center clustering. In Chapter 5, we generalize the definition of greedy permutations to achieve much wider classes of permutations, called approximate greedy permutations and locally greedy permutations. Then, we show how a net-tree can be constructed deterministically in linear time from such permutations. We also present algorithms to extract those permutations from a given net-tree.

Finally, in Chapter 6, we apply hierarchical trees to topological data analysis (TDA). In TDA, topological features of a given data set can be found using a sequence of growing balls centered at each point, called a filtration. A set of balls of the same radius can be represented combinatorially by a simplicial complex which consists of vertices, edges, triangles and their higher-dimensional analogues. The sequence of simplicial complexes corresponding to a filtration is called a simplicial filtration. A simplicial filtration can become very large very fast and it suffers from a combinatorial blow-up. We propose a sparsification technique that produces a linear size filtration that \((1 + \varepsilon)\)-approximates the original filtration. Then, we devise a data structure similar to net-trees to find a sparse filtration in linear time for doubling metrics.

1.1 Related Work

In this section, we review some important results in the metric data structure and topological data analysis literature.
1.1.1 Metric Structures

Metric data structures have been studied well in the literature. Uhlmann [66] proposed metric trees, also known as ball trees, to solve range searches in general metric spaces, but there are no performance guarantees on the queries. Yianilos [67] devised a similar data structure, called the vp-tree, and he showed that when the search radii are very small (the nearest neighbor distance is small), queries can be run in $O(\log n)$ expected time. These structures are balanced binary search trees and they can be constructed recursively in $O(n \log n)$ time by partitioning the points into two subsets according to their distance to the median.

Clarkson [23] proposed two randomized data structures to answer approximate nearest neighbor queries in metric spaces satisfying a sphere packing property, which is equivalent to having constant doubling dimension. The first data structure assumes that the query points have the same distribution as the given input points and it may fail to return a correct answer. The second one always returns a correct answer, but it requires more time and space. Roughly speaking, both of these data structures can be constructed in $O(n \log \Delta)$ time, with $O(n \log \Delta)$ size, and they answer queries in $O(\log \Delta)$ time.

Karger and Ruhl [50] proposed metric skip lists for the so-called growth restricted metrics, that are more restrictive than the doubling metrics. Their data structure is randomized and uses a Las Vegas algorithm. They showed that a metric skip list can be constructed dynamically in $O(n \log n \log \log n)$ time with $O(n \log n)$ size.

Krauthgamer and Lee [52] presented a dynamic deterministic data structure called navigating nets to address proximity searches in doubling spaces. Navigating nets are comprised of hierarchies of nested metric nets (see Section 2.2 for a definition) connected as a DAG. In more detail, the points at some scale $i$ are of distance $2^i$ and the balls centered at those points with radii $2^i$ contain all points of scale $i - 1$. 
They showed that navigating nets have linear size and can be constructed in $O(\log \Delta)$ time. They later [53] proposed algorithms to find approximate nearest neighbors in $\text{polylog}(n)$ using polynomial space under the blackbox model, where only oracle access to the distance function is given.

Gao et al. [33] independently devised a very similar data structure called the deformable spanner for Euclidean spaces. Their data structure is $(1 + \epsilon)$-spanner with $O(n/\epsilon^d)$ edges, and it supports dynamic insertions and deletions in $O(\log \Delta)$ time. Furthermore, they showed that a deformable spanner can be maintained under continuous motion.

Har-Peled and Mendel [41] developed net-trees and showed that a net-tree can be constructed in $O(n \log n)$ time independent of the spread. Their construction algorithm requires a complex sequence of approximating data structures. They used net-trees to efficiently solve many geometrical problems such as proximity searches, approximating the doubling dimension, etc..

Beygelzimer et al. [9] proposed the cover tree, a spanning tree of a navigating net, to make the space independent of the doubling dimension. Their experimental results showed that cover trees have good performance in practice. Besides the space complexity, cover trees do not theoretically outperform navigating nets and their construction time depends on the spread.

Cole and Gottlieb [24] proposed a data structure similar to net-trees that supports dynamic insertions and deletions in $O(\log n)$ time for points in doubling metrics. Their data structure maintains the so-called centroid path decomposition of a spanning tree of hierarchical metric nets. The centroid path decomposition partitions that spanning tree into a collection of paths that are each represented by a data structure called the biased skip list. Refer to [18, 22, 62] for complete surveys on proximity searches in metric spaces.
In this thesis, we propose a simple randomized incremental algorithm to construct net-trees in $O(n \log n)$ time in expectation and by doing some experiments, we evaluate its performance in practice. Furthermore, we prove that our construction algorithm only requires linear time if the given points are ordered in a greedy fashion.

1.1.2 Topological Data Analysis

Soon after the introduction of persistent homology by Edelsbrunner et al. [30], there was interest in building more elaborate complexes for larger and larger data sets. Following the full algebraic characterization of persistent homology by Zomorodian and Carlsson [69], a more general theory of zigzag persistence was developed [13, 14, 56, 57] using a more complicated algorithm.

Zigzags gave a way to analyze spaces that did not grow monotonically; they could alternately grow and shrink such as by growing the scale and then removing points [64]. A variant of this techniques was first applied for specific scales by Chazal and Oudot in work on manifold reconstruction [20] and was implemented as a full zigzag by Morozov in his Dionysus library [29]. Later, Sheehy gave a zigzag for Rips filtrations that came with guaranteed approximation to the persistent homology of the unsparsified filtration [63]. Other later works gave various improvements and generalizations of sparse zigzags.

Kerber and Sharathkumar generalized sparse Rips filtrations to Čech filtrations and related them to coresets of minimum enclosing balls [51]. Dey et al. gave an alternative construction of sparse Rips filtrations and a novel algorithm for simplicial map persistence using annotations [28]. Oudot and Sheehy introduced several Rips zigzags for homology inference from a sample with an emphasis on eliminating noise in the barcode at critical scales [58]. More recently, Botnan and Spreeman gave an alternative proof of the approximation ratio of sparse Rips filtrations using a novel

In this thesis, we propose a sparsification technique which simplifies the nerve filtration of a data set and we will show how the simplices can be extracted from the data set in linear time using a hierarchical structure.
Chapter 2

Preliminaries

In this chapter, we introduce some definitions and notations that will be used throughout this dissertation.

2.1 Distances and Metric Spaces

Let \((\mathcal{M}, d)\) be a metric space. The input is a finite subset \(P \subset \mathcal{M}\) with \(|P| = n\). For all \(p, q, r \in \mathcal{M}\), the distance function \(d\) satisfies the following properties:

1. non-negativity: \(d(p, q) \geq 0\),
2. self distance: \(d(p, p) = 0\),
3. symmetry: \(d(p, q) = d(q, p)\),
4. isolation: \(d(p, q) > 0 \iff p \neq q\),
5. triangle inequality: \(d(p, r) \leq d(p, q) + d(q, r)\).

The closed metric ball centered at \(p\) with radius \(r\) is denoted

\[ B(p, r) := \{q \in P \mid d(p, q) \leq r\}. \]
For a non-negative $\alpha \in \mathbb{R}$, the $\alpha$-offsets of $P$ are defined as

$$P^\alpha := \bigcup_{p \in P} B(p, \alpha).$$

The sequence of offsets as $\alpha$ ranges from 0 to $\infty$ is called the offsets filtration $\{P^\alpha\}$.

The doubling constant $\rho$ of $P$ is the minimum $\rho \in \mathbb{N}$ such that every ball $B(p, r)$ can be covered by $\rho$ balls of radius $r/2$. We assume $\rho$ is constant. The doubling dimension is defined as $\lg \rho$. A metric space with a constant doubling dimension is called a doubling metric. Throughout, we assume that the input metric is doubling. The doubling metric spaces are a robust family of metrics that appear in many practical settings.

The doubling dimension was first introduced by Assouad [6] and has since found many uses in algorithm design and analysis. Other notions of dimension for general metric spaces have also been proposed. The notion of growth-restricted metrics of Karger and Ruhl [50] is similar to doubling metrics, though it is more restrictive. Gupta et al. [37] showed that the doubling dimension of a growth restricted metric is an upper bound for its growth dimension.

Perhaps the most useful property of doubling metrics is that they allow for the use of packing and covering arguments similar to those used in Euclidean space to carry over to a more general class of metrics. The following lemma is at the heart of all the packing arguments in this dissertation.

**Lemma 2.1.1** (Packing Lemma). *If $X \subseteq B(p, r)$ and for every two distinct points $x, y \in X$, $d(x, y) > r'$, where $r > r'$, then $|X| \leq \rho^{\lfloor \lg(r/r') \rfloor + 1}$.***

Proof. By the definition of the doubling constant, $X$ can be covered by $\rho$ balls of radius $r/2$. These balls can each be covered by $\rho$ balls of radius $r/4$. Repeating this $\lfloor \lg(r/r') \rfloor + 1$ times results in at most $\rho^{\lfloor \lg(r/r') \rfloor + 1}$ balls of radius less than $r'$, and these balls contain at most one point of $X$ each, so $X$ has at most $\rho^{\lfloor \lg(r/r') \rfloor + 1}$ points. \qed
The spread of a point set $P$ often plays a role in running times of metric data structures. The following lemma captures the relationship between the spread and the cardinality of $P$ and follows directly from the Packing Lemma.

**Lemma 2.1.2.** A finite metric $P$ with doubling constant $\rho$ and spread $\Delta$ has at most $\rho^{O(\log \Delta)}$ points.

**Proof.** Without loss of generality, let the minimum pairwise distance in $P$ be 1. Then, the maximum pairwise distance is $\Delta$ and Lemma 2.1.1 proves the upper bound. \qed

The distance from a point $p$ to a compact set $Q$ is defined as

$$d(p, Q) := \min_{q \in Q} d(p, q).$$

The Hausdorff distance between two compact sets $P$ and $Q$ is

$$d_H(P, Q) := \max \left\{ \max_{p \in P} d(p, Q), \max_{q \in Q} d(q, P) \right\}.$$

### 2.2 Metric Nets and Net-Trees

An $(\alpha, \beta)$-net of a set $P$ is $Q \subset P$ such that

- **Packing:** for every two points $p, q \in Q$, $d(p, q) > \alpha$,

- **Covering:** for all $p \in P$, $d(p, Q) \leq \beta$.

Metric nets are important because they represent a given point set at a specific resolution. Har-Peled and Raichel [42] proposed a linear time algorithm to find nets in $\mathbb{R}^d$, which is an improvement over Har-Peled [39] which only works for low dimensional Euclidean spaces. They also used nets as bases to find linear time approximate solutions for many geometric problems such as $k$-center clustering and $k$-th smallest distance.
Figure 2.1: Nets at three different scales are shown from the left and the corresponding net-tree is illustrated on the right. White dots represent a net and the circles show the covering balls.

A net-tree is a tree of metric nets in which each level represents the metric space at some scale, see Fig. 2.1. In net-trees, points are leaves in level $-\infty$ and each point can be associated with many internal nodes. Each node is uniquely identified by its associated point and an integer called its level. The node in level $\ell$ associated with a point $p$ is denoted $p^\ell$. We assume that the root is in level $+\infty$. For a node $p^\ell \in T$, we define par($p^\ell$) and ch($p^\ell$) to be the parent and the set of children of that node, respectively. Let $P_{p^\ell}$ denote leaves of the subtree rooted at $p^\ell$. For each node $p^\ell$ in a net-tree, the following properties hold.

- **Packing:** $B(p, c_p \tau^\ell) \cap P \subseteq P_{p^\ell}$.

- **Covering:** $P_{p^\ell} \subset B(p, c_c \tau^\ell)$.

- **Nesting:** If $\ell > -\infty$, then $p^\ell$ has a child with the same associated point $p$.

The constant $\tau > 1$, called the *scale factor*, determines the change in scale between levels. We call $c_p$ and $c_c$ the *packing constant* and the *covering constant*, respectively, and $c_c \geq c_p > 0$. We represent all net-trees with the same scale factor, packing constant, and covering constant with $NT(\tau, c_p, c_c)$.

There are two different representations for net-trees, see Fig. 2.2. In the *uncompressed* representation, every root to leaf path has a node in every level. The size
complexity of this representation is $O(n \log \Delta)$, because there are $O(\log \Delta)$ explicit levels between $-\infty$ and $+\infty$. The compressed representation is obtained from the uncompressed one by removing the nodes that are the only child of their parents and they have only one child and merging the two adjacent edges as a long edge. We call such long edges jumps. It is not hard to see that this representation has size of $O(n)$. Note that the compressed net-trees are similar to compressed quadtrees.

A net-tree can be augmented to maintain a list of nearby nodes called relatives. We define relatives of a node $p^\ell$ to be

$$\text{rel}(p^\ell) := \{x^f \in T \mid y^g := \text{par}(x^f) \mid f \leq \ell < g, \text{ and } d(p, x) \leq c_r \tau^\ell\}.$$ 

We call $c_r$ the relative constant, and it is a function of the other parameters of a net-tree. In this dissertation, we assume that net-trees are always equipped with relatives.

Note that we defined ch(), par(), and rel() for a node of a tree; however, we abuse notation slightly and apply them to set of nodes. In such cases, the result will be the union of output for each node. Furthermore, the distance between nodes of a net-tree is the distance between their corresponding points.

Har-Peled and Mendel defined compressed net-trees in the class of $\text{NT}(11, \frac{\tau-5}{2(\tau-1)}; \frac{2\tau}{\tau-1})$ with $c_r = 13$. The following lemma uses the Packing Lemma and the definition of net-trees.

**Lemma 2.2.1.** For each node $p^\ell$ in $T \in \text{NT}(\tau, c_p, c_c)$, $|\text{ch}(p^\ell)| \leq \rho^{\lfloor \lg(c_c \tau/c_p) \rfloor + 1}$ and $|\text{rel}(p^\ell)| \leq \rho^{\lfloor \lg(c_r/c_p) \rfloor + 1}$.

**Proof.** By the covering property, $d(p^\ell, \text{ch}(p^\ell)) \leq c_c \tau^\ell$. Also, the packing property implies $d(p^\ell, \text{ch}(p^\ell)) > c_p \tau^{\ell-1}$. So, the Packing Lemma results $|\text{ch}(p^\ell)| \leq \rho^{\lfloor \lg(c_c \tau/c_p) \rfloor + 1}$. Furthermore, from the definition of relatives, $d(p^\ell, \text{rel}(p^\ell)) \leq c_r \tau^\ell$. Also, by the packing
Figure 2.2: Two representations of net-trees. Squares and ovals illustrate points and nodes of the trees, respectively. Horizontal dots show levels. Nodes with the same associated points are vertically aligned.

property, \( d(p^\ell, \text{rel}(p^\ell) \setminus \{p^\ell\}) > c_p \tau^\ell \). Thus, the Packing Lemma implies \( |\text{rel}(p^\ell)| \leq \rho^{\lfloor \log(c_r/c_p) \rfloor + 1}. \)

According to Lemma 2.2.1, a compressed net-tree on a doubling metric has \( \rho^{O(1)n} \) size.
2.3 Cover Trees

Cover trees are similar to net-trees, with the following properties:

- **Packing:** For all distinct \( p^\ell \) and \( q^\ell \), 
  \[ d(p, q) > c_p \tau^\ell. \]

- **Covering:** if \( q^m = \text{par}(p^\ell) \), then 
  \[ d(p, q) \leq c_c \tau^{\ell+1}. \]

- **Nesting:** If \( \ell > -\infty \), then \( p^\ell \) has a child with the same associated point \( p \).

Here, \( c_p \) and \( c_c \) are defined similar to net-trees. We denote all cover trees with the same parameters by \( \text{CT}(\tau, c_p, c_c) \). Beygelzimer et al. [9] defined cover trees in \( \text{CT}(2, 1, 1) \).

The main difference in the definitions of cover trees and net-trees is in the packing conditions. The net-tree requires the packing to be consistent with the hierarchical structure of the tree, a property not necessarily satisfied by the cover trees. Furthermore, cover trees do not maintain list of relatives which makes them true linear size data structures independent of the doubling dimension. Cover trees have simple algorithms and they work efficiently in practice [9].

2.4 Simplicial Complexes

A *simplicial complex* \( K \) is a family of subsets of a vertex set that is closed under taking subsets. The sets \( \sigma \in K \) are called *simplices* and \( |\sigma| - 1 \) is called the *dimension* of \( \sigma \). A nested family of simplicial complexes is called a *simplicial filtration*. Often the family of complexes will be parameterized by a nonnegative real number as in \( \{K^\alpha\}_{\alpha \geq 0} \). Here, the filtration property guarantees that \( \alpha \leq \beta \) implies that \( K^\alpha \subseteq K^\beta \). In this case, the value of \( \alpha \) for which a simplex first appears is called its birth time, and so, if there is a largest complex \( K^\alpha \) in the filtration, the whole filtration can be represented by \( K^\alpha \) and the birth time of each simplex. For this reason, simplicial filtrations are often called *filtered simplicial complex*. 
2.5 Persistent Homology

Homology is an algebraic tool for characterizing the connectivity of a space. It captures information about the connected components, holes, and voids. In this dissertation, we will only consider homology with field coefficients and the computations will all be on simplicial complexes. In this setting, computing homology is done by reducing a matrix \( D \) called the boundary matrix of the simplicial complex. The boundary matrix has one row and column for each simplex. If the matrix reduction respects the order of a filtration, i.e. columns are only combined with columns to their left, then the reduced matrix also represents the so-called persistent homology of the filtration. Persistent homology describes the changes in the homology as the filtration parameter changes and this information is often expressed in a barcode, see Fig 2.3. Barcodes give topological signatures of a shape [34].

![Figure 2.3: A filtration and its barcode.](image)

Each bar of a barcode is an interval encoding the lifespan of a topological feature in the filtration. We say that a barcode \( B_1 \) is a (multiplicative) \( c \)-approximation to another barcode \( B_2 \) if there is a partial matching between \( B_1 \) and \( B_2 \) such that every bar \([b, d]\) with \( d/b > c \) is matched and every matched pair of bars \([b, d], [b', d']\) satisfies
Figure 2.4: The nerve has an edge for each pairwise intersection, a triangle for each 3-way intersection (right), etc.

\[ \max\{\frac{b}{b'}, \frac{b'}{b}, \frac{d}{d'}, \frac{d'}{d}\} \leq c. \]

A standard result on the stability of barcodes [19] implies that if two filtrations \( \{F^\alpha\} \) and \( \{G^\alpha\} \) are \( c \)-interleaved in the sense that \( F^{\alpha/c} \subseteq G^\alpha \subseteq F^{c\alpha} \), then the barcode of \( \{F^\alpha\} \) is a \( c \)-approximation to \( \{G^\alpha\} \).

### 2.6 Nerve Complexes and Filtrations

Let \( \mathcal{U} = \{U_1, \ldots, U_n\} \) be a collection of closed, convex sets. Let \( \bigcup \mathcal{U} \) denote the union of the sets in \( \mathcal{U} \), i.e. \( \bigcup \mathcal{U} := \bigcup_{i=1}^n U_i \). We say that the set \( \mathcal{U} \) is a *cover* of the space \( \bigcup \mathcal{U} \). The *nerve* of \( \mathcal{U} \), denoted \( \text{Nrv}(\mathcal{U}) \) is the abstract simplicial complex defined as

\[
\text{Nrv}(\mathcal{U}) := \left\{ I \subseteq [n] \mid \bigcap_{i \in I} U_i \neq \emptyset \right\}.
\]

This construction is illustrated in Fig 2.4. The Nerve Theorem [43, Cor. 4G.3] implies that \( \text{Nrv}(\mathcal{U}) \) is homotopy equivalent to \( \bigcup \mathcal{U} \).

Similarly, one can construct a nerve filtration from a cover of a filtration by filtrations. Specifically, let \( \mathcal{U} = \{\{U_1^\alpha\}, \ldots, \{U_n^\alpha\}\} \) be a collection of filtrations parameterized by real numbers such that for each \( i \in [n] \) and each \( \alpha \geq 0 \), the set \( U_i^\alpha \) is closed and convex. As shorthand, we write \( \mathcal{U}^\alpha \) to denote the set \( \{U_1^\alpha, \ldots, U_n^\alpha\} \). As before, the Nerve
Theorem implies that $\bigcup U^\alpha$ is homotopy equivalent to $\text{Nrv}(U^\alpha)$. The Persistent Nerve Lemma [20] implies that the filtrations $\{\bigcup U^\alpha\}_{\alpha \geq 0}$ and $\{\text{Nrv}(U^\alpha)\}_{\alpha \geq 0}$ have identical persistent homology.

### 2.7 Čech and Rips Filtrations

A common filtered nerve is the Čech filtration. It is defined as $\{C^\alpha(P)\}$, where

$$C^\alpha(P) := \text{Nrv}\{B(p_i, \alpha) \mid i \in [n]\}.$$  

Notice that this is just the nerve of the cover of the $\alpha$-offsets by the $\alpha$-radius balls. Thus, the Persistent Nerve Lemma implies that $\{P^\alpha\}$ and $\{C^\alpha(P)\}$ have identical persistence barcodes.

A similar filtration that is defined for any metric is called the (Vietoris-)Rips filtration and is defined as $\{R^\alpha(P)\}$, where

$$R^\alpha(P) := \{J \subseteq [n] \mid \max_{i,j \in J} d(p_i, p_j) \leq 2\alpha\}.$$  

Note that if $d$ is the max-norm, $\ell_\infty$, then $R^\alpha(P) = C^\alpha(P)$. Moreover, because every finite metric can be isometrically embedded into $\ell_\infty$, every Rips filtration is isomorphic to a nerve filtration.
Chapter 3

Transforming Hierarchical Trees in Metric Spaces

In this chapter, we introduce different algorithms to transform hierarchical metric trees. In Section 3.1, we propose two modifications of net-trees which simplify our algorithms and analysis throughout this thesis. Then, in Section 3.2, we present a linear time algorithm to convert a cover tree to a net-tree. Finally, we introduce two linear time operations in Section 3.3 to establish a trade-off between the height and the node degree of net-trees.

3.1 Net-Tree Variants

In this section, we introduce two natural modifications of net-trees that simplify both construction and analysis. In the first variant, we replace the global packing and covering conditions of a net-tree with local ones that are easier to verify. We show that these local conditions imply the global conditions, thus they may be used in the

Some results in this chapter are published in [48].
analysis.

In the second variant, we show how a less aggressive compression criterion still results in a linear-size data structure while guaranteeing that relatives are on the same level in the tree, are symmetric, and are consistent up the tree (i.e. parents of relatives are relatives). This makes it much simpler to reason about local neighborhoods by local search among relatives.

3.1.1 Local Net-Trees

Here, we define a local version of net-trees and we show that for some appropriate parameters, a local net-tree is a net-tree. The “nets” in a net-tree are the subsets

\[ N_\ell := \{ p \in P \mid p^m \in T \text{ for some } m \geq \ell \}. \]

A local net-tree \( T \in \text{LNT}(\tau, c_p, c_c) \) satisfies the following invariants.

- **Local Packing:** For distinct \( p, q \in N_\ell \), \( d(p, q) > c_p \tau^\ell \).

- **Local Covering:** If \( q^m = \text{par}(p^\ell) \), then \( d(p, q) \leq c_c \tau^{\ell+1} \).

- **Local Parent:** If \( q^m = \text{par}(p^\ell) \), then \( d(p, q) = d(p, N_{\ell+1}) \).

- **Nesting:** If \( \ell > -\infty \), then \( p^\ell \) has a child with the same associated point \( p \).

The difference between the local net-tree invariants and the net-tree invariants given previously, is that there is no requirement that the packing or covering respect the tree structure. It is easy to see that the local packing and local covering properties can be obtained from the stronger ones. We are interested in local packing and covering properties because they are much easier to maintain as invariants after each update operation on a tree and also to verify in the analysis.
The switch to local net-trees comes at the cost of having slightly different constants. Theorem 3.1.2 gives the precise relationship. Its proof relies on the following lemma which will also be useful later. It bounds the distance of each node to its descendants in a local net-tree.

**Lemma 3.1.1.** For each descendant $x^\ell$ of $p^\ell$ in $T \in \text{LNT}(\tau,c_p,c_c)$, $d(p,x) < \frac{c_p\tau}{\tau - 1}\tau^\ell$

*Proof.* By the local covering property and the triangle inequality,

$$d(p,x) \leq \sum_{i=f+1}^{\ell} c_c\tau^i < c_c\frac{\tau}{\tau - 1}\tau^\ell.$$ 

**Theorem 3.1.2.** For $\tau > \frac{2c_c}{c_p} + 1$ and $0 < c_p \leq c_c < \frac{c_p(\tau - 1)}{2\tau - 1}$, if $T \in \text{LNT}(\tau,c_p,c_c)$, then $T \in \text{NT}(\tau,\frac{c_p(\tau - 1) - 2c_c}{2(\tau - 1)},\frac{c_c\tau}{\tau - 1})$.

*Proof.* The covering property can be proved using Lemma 3.1.1. To prove the packing property, let $p^\ell$ be a node of the local net-tree and $x \notin P_{p^\ell}$. Also, let $y^{\ell-1}$ and $z^\ell$ be the ancestors of $x$. From the parent property, $d(y,z) < d(y,p)$; otherwise, $p^\ell$ should be the parent of $y^{\ell-1}$. We have

$$d(p,z) \leq d(p,y) + d(y,z) \quad \text{[triangle inequality]}$$

\[
< 2d(p,y).
\]

So, $d(p,y) > d(p,z)/2$. Moreover,

$$d(p,x) \geq d(p,y) - d(y,x) \quad \text{[triangle inequality]}$$

\[
> \frac{1}{2} d(p,z) - d(y,x) \quad \text{[parent property]}
\]

\[
> \frac{1}{2} c_c\tau^\ell - \frac{c_c}{\tau - 1}\tau^\ell \quad \text{[Lemma 3.1.1]}
\]

\[
> \frac{c_p(\tau - 1) - 2c_c}{2(\tau - 1)}\tau^\ell.
\]

21
Therefore, \( P_{p^\ell} \subset B(p, \frac{c_p(\tau-1) - 2c_c}{2(\tau-1)} \tau^\ell) \).

If \( c_c = c_p = 1 \), then a local net-tree with \( \tau > 3 \) belongs to \( NT(\tau, \frac{\tau-3}{2(\tau-1)}, \frac{\tau}{\tau-1}) \), which results in a definition of net-trees similar to Har-Peled and Mendel’s [41].

In the rest of this dissertation, we focus on local net-trees and our algorithms construct such trees. Theorem 3.1.2 implies that the output is also a net-tree albeit with different parameters. Furthermore, we ignore the prefix of local when we refer to the local net-tree properties.

### 3.1.2 Semi-Compressed Net-Trees

In this section, we define *semi-compressed* net-trees. This intermediate structure between uncompressed and compressed net-trees has linear size, and produces a neighborhood graph that is easier to work with because edges are undirected and stay on the same level of the tree. As we will see later, this representation greatly simplifies the algorithms and analysis.

Recall that for compressed net-trees, we remove a node (by compressing edges) if it is the only child of its parent and has only one child. In semi-compressed net-trees, we do not remove a node if it has any relatives other than itself. Figure 3.1 illustrates different representations of a net-tree on a set of points on a line. In the following theorem, we show that the semi-compressed representation has linear size.

**Theorem 3.1.3.** Given \( n \) points \( P \) in a doubling metric with doubling constant \( \rho \). The size of a semi-compressed net-tree on \( P \) is \( O(\rho^{\lg(c_{c}/c_{p})} n) \).

**Proof.** Let \( T \) be an uncompressed net-tree on \( P \). Let \( S \) be the semi-compressed tree formed from \( T \). That is, \( S \) contains (exactly) the root of \( T \) and every \( p^\ell \in T \) such that \( |\text{rel}(p^\ell)| > 1 \). It should be clear that the size of \( S \) is \( O(n + m) \) where \( n \) is the number
Figure 3.1: Different representations of a net-tree $T \in \text{NT}(2,1,1)$ with $c_r = 4$ on a set of points $\{0, 2, 11, 28\}$ on a line. An arrow from node $a$ to node $b$ indicates $b$ is a relative of $a$. By definition, each node is its own relative; however, the figure does not show such relationship.

of input points and $m$ is the number of relative edges in the whole tree. So, if we can show $m = O(n)$, we will have shown that $S$ has linear size.

First, we show that if two points $p$ and $q$ are relatives at some level $\ell$ in $T$, then they can be relatives in at most $\lceil \log_\tau (c_r/c_p) \rceil$ levels. Without loss of generality, let $\ell - i$ be
the lowest level in $T$ such that $p$ and $q$ are relatives, where $i \geq 0$. Then, $d(p, q) \leq c_r \tau^{\ell-i}$.

By the packing property at level $\ell$, $d(p, q) > c_p \tau^\ell$. Combining the last two inequalities results $c_p \tau^\ell < c_r \tau^{\ell-i}$. Therefore, $p$ and $q$ are relatives in $i \leq \lceil \log_\tau(c_r/c_p) \rceil$ levels.

Now, we find the number of relative edges. For a point $p$, we define

$$h(p) := \max\{\ell \mid p^\ell \in T\}.$$ 

If $p$ and $q$ are relatives in $T$, then we charge the point having $\min\{h(p), h(q)\}$ with the cost of $O(\log_\tau(c_r/c_p))$ to pay for the total number of relative edges between $p$ and $q$. Therefore, using Lemma 2.2.1

$$m = O\left(\sum_{q \in P} O\left(\log_\tau(c_r/c_p)\right) |\text{rel}(q^{h(q)})|\right) = O(\rho^{\lg(c_r/c_p)} n). \quad \square$$

A semi-compressed representation can be obtained from both uncompressed and compressed representations. To convert an uncompressed representation to semi-compressed, one needs to iterate over all nodes in the tree and remove extra nodes according to the aforementioned condition. This operation takes $O(n \log \Delta)$ time. The conversion from compressed representation to semi-compressed is another useful operation for net-trees. In the following theorem, we show that this operation can be done in linear time.

**Theorem 3.1.4.** Given a compressed net-tree $T \in \text{LNT}(\tau, c_p, c_c)$ with relative constant $c_r$. One can change the representation of $T$ to semi-compressed in $O(\rho^{\lg(c_r/c_p)} n)$ time.

**Proof.** To obtain a semi-compressed representation from $T$, we need to split some jumps at appropriate levels where necessary. We categorize all pairs of relatives $(p^\ell, q^m)$, where $q^m \in \text{rel}(p^\ell)$, into three groups: both $p^\ell$ and $q^m$ are parts of jumps (top or bottom), only one of them is a part of a jump, and none of them are parts of jumps. There is
no need to take any action for the last group. However, the algorithm first handles the first group and then considers the second group. In the following, we consider each group separately.

**Group 1:** Both $p$ and $q^m$ are parts of jumps. We have four possibilities for all the pairs in this group.

(a) *Both $p^\ell$ and $q^m$ are the top of jumps.* In this case, we cannot have $m < \ell$. Because by the definition of relatives, $q^m$ should be the bottom of another jump. From the definition of compressed hierarchical trees, a node cannot serve simultaneously as the top and the bottom of jumps. So, $m = \ell$. Now, we require the minimum value of $i$ such that $p$ and $q$ are still relatives in level $\ell - i$. By the packing property in level $\ell$ and the relative property in level $\ell - i$,

$$c_p \tau^\ell < d(p, q) \leq c_r \tau^{\ell - i}.$$  

So, $i < \log_r(c_r/c_p)$ and it means that $p$ and $q$ are relatives in constant number of levels.

(b) *$p^\ell$ is the top and $q^m$ is the bottom of jumps.* If $m = \ell$, then by the definition of compressed cover trees $p^\ell$ and $q^m$ cannot be the bottom and the top of other jumps, respectively. So, such pairs are not contributing in the split of the corresponding jumps and the algorithm simply ignores these pairs. If $m < \ell$, then we create a new node $q^\ell$ and this node is a relative of $p^\ell$. We can handle it similar to case (a) with pair $p^\ell$ and $q^\ell$.

(c) *$p^\ell$ is the bottom and $q^m$ is the top of jumps.* Using the same argument of case (a), it is impossible to have $m < \ell$. Also, the algorithm ignores such pairs if $m = \ell$, similar to case (b).
(d) Both \( p^\ell \) and \( q^m \) are the bottom of jumps. If \( m = \ell \), then \( p \) and \( q \) are relatives up to some level \( \ell + i \). By the packing property, \( d(p^{\ell+i}, q^{\ell+i}) > c_p \tau^{\ell+i} \). Therefore, \( c_p \tau^{\ell+i} < c_r \tau^\ell \) results \( i < \log_r(c_r/c_p) \). For case \( m < \ell \), first we create node \( q^\ell \), then the rest is similar to \( m = \ell \).

**Group 2:** Only one of \( p^\ell \) and \( q^\ell \) is a part of a jump. If \( p^\ell \) is the top or the bottom of a jump and \( q^m \) is not part of a jump, then \( p^\ell \in \text{rel}(q^m) \) and no further action is required. Otherwise, \( p^\ell \) is neither the top nor the bottom of a jump and \( q^m \) is the bottom of a jump. In this case, we only need to create a new node \( q^\ell \).

Lemma 2.2.1 implies that we require \( O(\log_r(c_r/c_p)\rho^{\log(c_r/c_p)}n) \) time to handle all the pairs of the first group. Note that the creation of \( q^\ell \) in the second group does not result cascading insertions, because all pairs of relative jumps are handled by the first group and \( q^\ell \) does not have a jump node in its relative list. So, the algorithm only requires \( O(\rho^{\log(c_r/c_p)}n) \) time for the second group. Therefore, \( T \) can be converted to a semi-compressed local net-tree in \( O(\rho^{\log(c_r/c_p)}n) \) time.

**Lemma 3.1.5.** In semi-compressed net-trees, the relative relation is symmetric, i.e. if \( p^\ell \in \text{rel}(q^\ell) \) then \( q^\ell \in \text{rel}(p^\ell) \).

*Proof.* Let \( x^m := \text{par}(q^\ell) \). From the definition of relatives \( q^\ell \in \text{rel}(p^\ell) \) if and only if \( \ell < m \) and \( d(p,q) \leq c_r \tau^\ell \). The first condition trivially holds. The latter also holds because \( p^\ell \in \text{rel}(q^\ell) \).

From now on, we use \( \sim \) to denote the symmetric relative relation between pairs of nodes in semi-compressed net-trees. In the following lemma, we prove if two nodes are relatives, then their parents are also relatives.

**Lemma 3.1.6.** In a semi-compressed net-tree \( T \in \text{LNT}(\tau, c_p, c_c) \) with \( c_r \geq 2c_c \tau/(\tau-1) \), if \( p^\ell \sim q^\ell \), then \( \text{par}(p^\ell) \sim \text{par}(q^\ell) \).
Proof. By applying the triangle inequality, the covering, and the relative properties

\[
d(\text{par}(p^\ell), \text{par}(q^\ell)) \leq d(\text{par}(p^\ell), p^\ell) + d(p^\ell, q^\ell) + d(q^\ell, \text{par}(q^\ell))
\]

\[
\leq c_c \tau^{\ell+1} + c_r \tau^\ell + c_c \tau^{\ell+1}
\]

\[
\leq c_r \tau^{\ell+1}.
\]

\[\square\]

### 3.2 Cover Trees to Net-Trees

Cover trees are simple data structures that can be built efficiently in practice. On the other hand, net-trees provide better theoretical guarantees but they come at the cost of a complex preprocessing step. In this section, we show how a net-tree can be built from a given cover tree.

In Section 3.1, we proposed local net-trees and showed that they satisfy global properties of net-trees with slightly different parameters. The main difference of local net-trees and cover trees is that local net-trees satisfy the parent invariant and they also maintain lists of relatives. Here, we propose an algorithm that does this conversion by augmenting the cover tree with relatives and enforcing the parent condition. Algorithm 1 converts a given cover tree to a local net-tree. Note that the outer loop of Algorithm 1 only iterates over all explicit (non-empty) levels of the cover tree.

**Theorem 3.2.1.** Algorithm 1 correctly converts a(n) (un)compressed cover tree \( T \in \text{CT}(\tau, c_p, c_c) \) to a(n) (un)compressed local net-tree \( T' \in \text{LNT}(\tau > 1, c_p, c_c) \) with \( c_r \geq \max\{2c_c, \frac{c_c \tau^2}{(\tau - 1)^2}\} \).

**Proof.** We need to show that both the parent and the relatives are correctly found for each node of a tree. If the parent of \( p^\ell \) is a node of \( p \), then the parent property holds trivially. Otherwise, the parent of \( p^\ell \) should be in level \( \ell + 1 \). By the covering property, \( d(p^\ell, \text{par}(p^\ell)) \leq c_c \tau^{\ell+1} \). Let \( x^\ell \) be the lowest and the closest node to \( p^\ell \) among all
Algorithm 1 Convert a cover tree to a local net-tree

1: procedure CoverTreeToNetTree($T, c_r$)
2: for all explicit levels $\ell$ from $+\infty$ to $-\infty$ do
3: for all $p^\ell \in T$ do
4: rel($p^\ell$) $\leftarrow \{p^\ell\}$
5: if $\ell < +\infty$ then
6: $q^m \leftarrow$ par($p^\ell$)
7: FindParent($p^\ell$, false)
8: candidates $\leftarrow$ rel($q^m$) $\cup$ ch(rel($q^m$)) $\setminus \{p^\ell\}$
9: for all $x^f \in$ candidates do
10: $y^g \leftarrow$ par($x^f$)
11: if $f \leq \ell < g$ and $d(p, x) \leq c_r \tau^\ell$ then
12: Add $x^f$ to rel($p^\ell$)
13: else if $\ell \leq f < m$ and $d(p, x) \leq c_r \tau^f$ then
14: Add $p^\ell$ to rel($x^f$)
15: Add ch($x^f$) to candidates

nodes with levels greater than $\ell$. Then $d(p, x) \leq d(p, q)$. We have

\[
\begin{align*}
    d(q, x) &\leq d(p, q) + d(q, x) & \text{[triangle inequality]} \\
    &\leq 2d(p, q) & \text{[parent property]} \\
    &\leq 2c_r \tau^{\ell+1} & \text{[covering property]} \\
    &\leq c_r \tau^{\ell+1}. & \text{[for } c_r \geq 2c_c]\end{align*}
\]

Therefore, either $f = \ell + 1$ or the only child of $x^f$ is in level $\ell$, because otherwise $x^f$ is not the lowest which contradicts with the assumption. Thus, either $x^f$ or its only child belongs to rel($q^m$).

Now, we prove that relatives are correct with $c_r \geq \frac{c_c \tau^2}{(\tau-1)^2}$. Suppose for a contradic-
Algorithm 2 Ensure that the parent of \( p^\ell \) satisfies the parent property

1: procedure FindParent\((p^\ell, \text{semi})\)
2: \( q^m \leftarrow \text{par}(p^\ell) \)
3: Let \( x^f \) be the closest node to \( p^\ell \) in \( \text{rel}(q^m) \)
4: if \( x \neq q \) then
5: if \( f < \ell + 1 \) then
6: \( x^f \leftarrow \text{par}(x^f) \)
7: if \( f > \ell + 1 \) then
8: Split the jump at \( x^f \) at levels \( \ell + 1 \) and \( \ell \)
9: Add \( p^\ell \) to \( \text{ch}(x^{\ell+1}) \)
10: if \( |\text{ch}(q^m)| = 1 \) and \( |\text{ch}(\text{ch}(q^m))| = 1 \) then
11: \( q^\ell \leftarrow \text{ch}(q^m) \)
12: if \( \text{semi} = \text{false} \) or \( \text{semi} = \text{true} \) and \( |\text{rel}(q^\ell)| = 1 \) then
13: \( \text{ch}(q^m) \leftarrow \text{ch}(q^\ell) \)
14: Delete \( q^\ell \)

Case 1: \( x^f \notin \text{rel}(q^m) \). In this case, at least one of the two conditions of relatives does not hold for \( x^f \). If \( d(q, x) > \frac{c_\epsilon \tau^2}{(\tau - 1)^2} \tau^m \), then

\[
\begin{align*}
d(p, x) & \geq d(q, x) - d(p, q) \\
& > \frac{c_\epsilon \tau^2}{(\tau - 1)^2} \tau^m - c_\epsilon \tau^m \\
& \geq c_\epsilon \frac{2\tau - 1}{(\tau - 1)^2} \tau^{\ell + 1}. 
\end{align*}
\]

[triangle inequality]
[covering property]

[\( m \geq \ell + 1 \)]
We assumed that \( x^f \in \text{rel}(p^f) \), so \( d(p, x) \leq \frac{c_\epsilon \tau^2}{(\tau-1)^2} \tau^f \). These inequalities imply \( \tau < 1 \), a contradiction. If \( d(q, x) \leq \frac{c_\epsilon \tau^2}{(\tau-1)^2} \tau^m \) and \( g > f > m \), then \( f > \ell \) is also a contradiction. The last case \( d(q, x) \leq \frac{c_\epsilon \tau^2}{(\tau-1)^2} \tau^m \) and \( f < g \leq m \) is a special case of \( p^f \notin \text{rel}(s^h) \), which is described in the following.

**Case 2:** \( p^f \notin \text{rel}(s^h) \). In this case, \( \ell < h < m \) and \( d(p, s) > \frac{c_\epsilon \tau^2}{(\tau-1)^2} \tau^h \). Then,

\[
d(p, s) \leq d(p, x) + d(x, s) \quad \text{[triangle inequality]}
\]

\[
< \frac{c_\epsilon \tau^2}{(\tau-1)^2} \tau^\ell + \frac{c_\epsilon \tau}{\tau-1} \tau^h \\
\leq \frac{c_\epsilon \tau^2}{(\tau-1)^2} \tau^{h-1} + \frac{c_\epsilon \tau}{\tau-1} \tau^h \\
= \frac{c_\epsilon \tau^2}{(\tau-1)^2} \tau^h.
\]

This is a contradiction because \( d(p, s) > \frac{c_\epsilon \tau^2}{(\tau-1)^2} \tau^h \).

**Theorem 3.2.2.** Algorithm 1 converts a compressed cover tree to a compressed local net-tree in \( O(\rho^{\lg(c_\epsilon \epsilon \tau/c_p)} n) \) time.

**Proof.** The two outermost loops run as many as the number of nodes in a cover tree in total, which is \( O(n) \). By Lemma 2.2.1, finding the right parent requires \( O(\rho^{\lg(c_\epsilon \epsilon \tau/c_p)}) \). Therefore, the algorithm enforces the parent invariant in \( O(\rho^{\lg(c_\epsilon \epsilon \tau/c_p)} n) \) time. We use an amortized analysis for the time complexity of finding relatives.

When two nodes are checked against the relative condition, either it succeeds or fails. If it succeeds, we charge the node whose relative list has been grown. By Lemma 2.2.1, each node needs \( O(\rho^{\lg(c_\epsilon \epsilon \tau/c_p)}) \) charge for this case. Otherwise, we charge one of the two nodes to pay the cost of a failed check in the following way. If \( x^f \in \text{rel}(q^m) \cup \text{ch}(\text{rel}(q^m)) \), then \( p^f \) pays the cost of a failed check. Using Lemma 2.2.1, each node requires \( O(\rho^{\lg(c_\epsilon \epsilon \tau/c_p)}) \) charge. Otherwise, \( p^f \in \text{rel}(\text{par}(x^f)) \) and we charge
par\( (x') \) to pay the cost of a failed check. Using Lemma 2.2.1, each node requires 
\( O(\rho^g(c_p/c_r^2)) \) charge for failed checks. By Theorem 3.1.3, the output has 
\( O(\rho^g(c_p/c_r)n) \) nodes. Therefore, the total time complexity is 
\( O(\rho^g(c_p/c_r^2)n) \).

\[ \square \]

## 3.3 Transforming Net-Trees

For a net-tree, there is a trade-off between the height of the tree and the scale factor. 
It is not hard to see that the height of a net-tree can be \( O(\log_\tau \Delta) \). So by increasing 
the scale factor, the height of the tree will be decreased. Also, from Lemma 2.2.1, 
increasing the scale factor results in more children for each node of a net-tree. In this 
section, we define two operations to change the scale factor of a given semi-compressed 
local net-tree. A *coarsening* operation modifies the tree to increase the scale factor. 
Similarly, a *refining* operation results a tree with smaller scale factor.

Note that in Theorem 3.1.2, we proved that any local net-tree with \( \tau > \frac{2c_p}{c_r} + 1 \) is also a (global) net-tree. However, in some cases we may have \( \tau \leq \frac{2c_p}{c_r} + 1 \) and require 
stronger conditions of net-trees. Beygelzimer et. al. [9] set \( \tau = 2 \) in their definition of 
cover trees and found that \( \tau = 1.3 \) is even more efficient in practice. For such cover 
trees, Algorithm 1 only yields a local net-tree which is not a (global) net-tree. In these 
cases, we can use the coarsening operation so that Theorem 3.1.2 can be applied on 
the resulted local net-tree.

### 3.3.1 Coarsening

Here, we convert a semi-compressed local net tree \( T \) with the scale factor of \( \tau \) to 
another semi-compressed local net-tree \( T' \) with the scale factor of \( \tau' := \tau^k \), for some 
constant integer \( k > 1 \). The coarsening operation can be seen as combining every \( k \) 
levels of \( T \) into one level in \( T' \). We define a mapping between nodes of \( T \) and \( T' \) as
follows. In this mapping, each node $p^\ell$ in $T$ maps to a node $p'^\ell = p^{[\ell/k]}$ in $T'$. Here, we use prime as a function that indicates the level of a node in $T'$ that corresponds to $p^\ell$, i.e. $\ell' := [\ell/k]$. We also assume that each point $p$ in $T'$ maintains high($p, T'$), which is the highest node of $T'$ associated to point $p$. More formally,

$$
\text{high}(p, T) := \arg \max_{q^h \in T} \{ h | q = p \}.
$$

Algorithm 3 describes the coarsening operation. In this algorithm, we build the coarser tree from the bottom up, which implies that in the middle of the operation we have a set of disjoint trees. This algorithm then invokes \textsc{CoverTreeToNetTree} procedure to enforce the parent condition and augment $T'$ with relatives. Note that $T'$ is not exactly compressed or uncompressed. Finally, \textsc{MakeSemiCompressed} is used to ensure that $T'$ has a semi-compressed representation. \textsc{MakeSemiCompressed} first uses the algorithm of Theorem 3.1.4 to create those nodes that are required to exist in the semi-compressed representation, and then iterates over all nodes and removes those extra nodes violating the semi-compressed condition.

**Theorem 3.3.1.** Given a semi-compressed tree $T \in \text{LNT}(\tau, c_p, c_c)$ with $c_r \geq \frac{2c_0 \tau}{\tau - 1}$, Algorithm 3 converts $T$ to a semi-compressed tree $T' \in \text{LNT}(\tau', c_p', c_c', c_r')$ with $c_r' \geq c_r$, where $k > 1$ is a constant integer.

**Proof.** The theorem requires showing the three invariants including the covering, the packing, and the parent hold and the output is semi-compressed. First we prove that $T'$ satisfies the covering property. If $m > k([\ell/k] + 1)$, then $q^m$ is the top of a jump with the bottom node in a level less that $k([\ell/k] + 1)$, where Lemma 3.1.1 implies $d(q, p) \leq \frac{c_r \tau}{\tau - 1} k([\ell/k] + 1) - 1$. Otherwise, using Lemma 3.1.1, we have $d(q, p) \leq \ldots$
Algorithm 3 Coarsening operation for a given semi-compressed local net-tree

1: procedure Coarsening($T \in \text{LNT}(\tau, c_p, c_c), k, c'_r)$
2: Initialize $T' \in \text{LNT}(\tau' := \tau^k, c'_p := c_p, c'_c := \frac{c_c}{\tau^k})$ to be an empty local net-tree
3: for all explicit levels $\ell$ in $T$ from $-\infty$ to $+\infty$ do
4: for all $p^\ell \in T$ do
5: if $\ell = -\infty$ then
6: Create node $p^{-\infty}$ in $T'$
7: high($p, T'$) $\leftarrow$ $p^{-\infty}$
8: else
9: $q^m \leftarrow$ the lowest ancestor of $p^\ell$ with $m \geq k([\ell/k] + 1)$
10: $q'^\ell \leftarrow$ high($q, T'$)
11: if $f' < \ell'$ then
12: Create node $q'^\ell$ in $T'$ and add $q'^\ell$ to ch($q'^\ell$)
13: high($q, T'$) $\leftarrow$ $q'^\ell$
14: $q'^\ell \leftarrow$ high($q, T'$)
15: if $f' = \ell'$ then
16: Create node $q'^{\ell'+1}$ in $T'$ and add $q'^\ell$ to ch($q'^{\ell'+1}$)
17: high($q, T'$) $\leftarrow$ $q'^{\ell'+1}$
18: if $p \neq q$ then
19: $p'^h \leftarrow$ high($p, T'$)
20: if $h' < \ell'$ then
21: Create node $p'^\ell$ in $T'$ and add $p'^h$ to ch($p'^\ell$)
22: high($p, T'$) $\leftarrow$ $p'^\ell$
23: Add $p'^\ell$ to ch($q'^{\ell'+1}$)
24: COVERTREETOTREETREE($T'$, $c'_r$)
25: MAKESEMICOMPRESSED($T'$)
26: return $T'$

33
\[
\frac{c_c \tau}{\tau - 1} k(\ell/k + 1),
\]
Therefore,
\[
d(p, q) < \frac{c_c \tau}{\tau - 1} \tau^{k(\ell/k + 1)} < \frac{c_c \tau}{\tau - 1} (\tau')^{\ell' + 1} = c'_c (\tau')^{\ell' + 1}.
\]

It follows that \( T' \) has covering constant \( \frac{c_c \tau}{\tau - 1} \).

Next, we prove that the packing constant is correct. For every two nodes \( p^\ell, q^\ell \in T \),
\[
d(p, q) > c_p \tau^\ell = c_p \tau^{k_{\ell/k}} \geq c_p \tau^{k_{\ell/k}} = c_p (\tau')^{\ell'}.
\]
Thus, the points in level \( \ell' \) of \( T' \) satisfy the packing condition with \( c'_p = c_p \).

Theorem 3.2.1 implies that \( \text{COVERTREE} \to \text{NetTree} \) correctly finds the parent and relatives of every node in \( T' \). Also, it is easy to see that \( \text{MAKESEMI} \to \text{COMPRESSED} \) correctly turns \( T' \) into semi-compressed representation.

**Theorem 3.3.2.** The time complexity of Algorithm 3 is \( O(\rho^{\log(c'^2 c'_c \tau'/c'_p)} n) \).

**Proof.** As the preprocessing step, we can maintain the lowest ancestor of all nodes in \( T \) in \( O(n) \) time, which supports a constant time access to each ancestor [7]. Before calling \( \text{COVERTREE} \to \text{NetTree} \), every node \( p^\ell \in T \) may create at most three nodes in \( T' \), so \( T' \) has \( O(\rho^{\log(c'/c_p)} n) \) size by Theorem 3.1.3. From the proof of Theorem 3.2.2, \( \text{COVERTREE} \to \text{NetTree} \) requires \( O(\rho^{\log(c'^2 c'_c \tau'/c'_p)} n) \) time and the resulted tree has \( O(\rho^{\log(c'/c_p)} n) \) + \( O(\rho^{\log(c'/c_p)} n) = O(\rho^{\log(c'/c_p)} n) \) nodes. Using Theorem 3.1.4, \( \text{MAKESEMI} \to \text{COMPRESSED} \) needs \( O(\rho^{\log(c'/c_p)} n) \) time to restore the removed nodes required in the semi-compressed representation. Then, \( \text{MAKESEMI} \to \text{COMPRESSED} \) spends \( O(\rho^{\log(c'/c_p)} n) \) time to remove extra nodes. Therefore, the total running time of \( \text{COARSENING} \) is \( O(\rho^{\log(c'^2 c'_c \tau'/c'_p)} n) \).
3.3.2 Refining

Decreasing the scale factor is another useful operation for net-trees, and we call this operation refining. To refine a given semi-compressed local net-tree \( T \), each level \( \ell \) in \( T \) is split into at most \( k \) levels \( k\ell, \ldots, (k\ell + k - 1) \) in \( T' \). As a result, a node \( p^{\ell} \) in \( T \) may appear at most in \( k \) levels \( k\ell, \ldots, (k\ell + k - 1) \) of \( T' \). Similar to the coarsening operation, for each point \( p \) in \( T' \) we maintain \( \text{high}(p, T') \).

Algorithm 4 describes the refining operation. This algorithm uses \text{AddNode}, which searches downward from a node \( p^{\ell} \) to find a node of \( p \) at some given level \( h \). If node \( p^h \) does not exist in the tree, it will be added to the tree. To ensure that the result has a semi-compressed representation, we use \text{MakeSemiCompressed} introduced in Section 3.3.1.

**Theorem 3.3.3.** Given a semi-compressed tree \( T \in \text{LNT}(\tau, c_p, c_c) \) with \( c_r \geq \frac{2c_r\tau}{\tau - 1} \). Algorithm 3 converts \( T \) to a semi-compressed tree \( T' \in \text{LNT}(\tau' := \frac{\tau}{k}, c_p' := c_p, c_c' := c_c) \) with \( c_r' \geq c_r \), where \( k > 1 \) is a constant integer.

**Proof.** First we show that the packing and the covering properties are maintained. Note that the parent found in the for loop does not necessarily satisfy the parent property in \( T' \), because there may exist a node \( x^{\ell} \in T \) closer to \( p \) than the current parent of \( p \) in \( T' \) and processed after \( p^{\ell} \). In this case, since \( x^{\ell} \) is not processed yet, \( \text{high}(x, T') \) cannot cover \( p \). However, finding the parent helps us to maintain the packing property. By the packing property, for every \( x^{\ell} \in T \),

\[
\text{d}(p, x) > c_p\tau^{\ell} = c_p\tau^{k\ell/k} = c_p(\tau')^{k\ell},
\]

which implies that the insertion of \( p \) at any level between \( k\ell + 1 \) to \( k(\ell + 1) - 1 \) may violate the packing property. Therefore, finding the closest covering node among nodes
Algorithm 4 Refining operation for a given semi-compressed local net-tree

1: **procedure** Refining($T \in \text{LNT}$(τ, cp, cc), k, cr)
2: \hspace{1em} Initialize $T' \in \text{LNT}(\tau' := \tau^{1/k}, c_p' := c_p, c_c' := c_c)$ to be an empty local net-tree
3: \hspace{1em} for all explicit levels $\ell$ in $T$ from $-\infty$ to $+\infty$ do
4: \hspace{2em} for all $p^\ell \in T$ do
5: \hspace{3em} if $\ell = -\infty$ then
6: \hspace{4em} Create $p^{-\infty}$ in $T'$
7: \hspace{4em} high($p, T'$) ← $p^{-\infty}$
8: \hspace{3em} else
9: \hspace{4em} $q^m$ ← par($p^\ell$)
10: \hspace{4em} $p^{h'}$ ← high($p, T'$)
11: \hspace{4em} if $p = q$ and $h' < k\ell$ then
12: \hspace{5em} AddNode($T', \text{high}(p, T'), k\ell + 1$)
13: \hspace{5em} AddNode($T', \text{high}(p, T'), k\ell$)
14: \hspace{3em} else if $p \neq q$ then
15: \hspace{4em} $x^f$ ← $q^m$
16: \hspace{4em} for all $y^g \in \text{ch(rel}(q^m))$ do
17: \hspace{5em} $y^g'$ ← high($y, T'$)
18: \hspace{5em} if $d(p, y) < d(p, x)$ and $d(p, y) \leq c_c'_{p}(\tau')^{g'}$ then
19: \hspace{6em} $x^f$ ← $y^g$
20: \hspace{4em} Find $h'$ such that $c_p'(\tau')^{h'-1} < d(p, x) \leq c_c'(\tau')^{h'}$
21: \hspace{4em} $x^{h'}$ ← AddNode($T', \text{high}(x, T'), h'$)
22: \hspace{4em} $x^{h'-1}$ ← AddNode($T', x^{h'}, h' - 1$)
23: \hspace{4em} $p^{h'-1}$ ← AddNode($T', \text{high}(p, T'), h' - 1$)
24: \hspace{4em} Add $p^{h'-1}$ as a child of $x^{h'}$
25: \hspace{1em} CoverTreeToNetTree($T'$, $c'_r$)
26: \hspace{1em} MakeSemiCompressed($T'$)
27: \hspace{1em} return $T'$
Algorithm 5 Return a node of \( p \) at level \( h \) by descending the tree from a given node \( p^\ell \):

1: \textbf{procedure} \textsc{AddNode}(\( T, p^\ell, h \))
2: \hspace{1em} \textbf{if} \( h = \ell \) \textbf{then}
3: \hspace{2em} \textbf{return} \( p^\ell \)
4: \hspace{1em} \textbf{else if} \( h > \ell \) \textbf{then}
5: \hspace{2em} Create \( p^h \) and add \( p^\ell \) to \( \text{ch}(p^h) \)
6: \hspace{2em} high\((p, T) \leftarrow p^h \)
7: \hspace{2em} \textbf{return} \( p^h \)
8: \hspace{1em} \textbf{next} \leftarrow p^\ell
9: \hspace{1em} \textbf{while} \( p^h \notin \text{ch(\text{\textbf{next})}} \) \textbf{do}
10: \hspace{2em} Let \( p^f \) be a child of \( \text{\textbf{next}} \)
11: \hspace{2em} \textbf{if} \( f < h \) \textbf{then}
12: \hspace{3em} Create \( p^h \), add \( p^f \) to \( \text{ch}(p^h) \) and add \( p^h \) to \( \text{ch(\text{\textbf{next})}} \)
13: \hspace{2em} \textbf{else}
14: \hspace{3em} \text{\textbf{next}} \leftarrow p^f
15: \hspace{1em} \textbf{return} \( \text{\textbf{next}} \)

nearby \( p^\ell \) in Line 18 not only ensures the covering property of \( T' \), but also helps us to find a level in Line 20 to satisfy the packing property.

Theorem 3.2.1 ensures that the algorithm correctly finds parents and relatives. Furthermore, \textsc{MakeSemiCompressed} procedure turns \( T' \) into a correct semi-compressed representation.

\textbf{Theorem 3.3.4.} Algorithm 4 runs in \( O(k\rho^{\log(c_2^2 c_\tau/c_3^3)} n) \) time.

\textit{Proof.} Since each level of \( T \) can be split into \( k \) levels in \( T' \), \textsc{AddNode} runs in \( O(k) \), and by Theorem 3.1.3, \( T' \) has \( O(k\rho^{\log(c_2/c_3)} n) \) nodes before the execution of \textsc{CoverTree-ToNetTree}. Using Lemma 2.2.1, the time complexity of finding a parent from \( \text{ch(\text{rel(\text{\textbf{q}}^m))}} \) is \( O(\rho^{\log(c_2 c_\tau/c_3^3)}) \), so the algorithm requires \( O(k\rho^{\log(c_2^2 c_\tau/c_3^3)} n) \) time to finish.
In Theorem 3.2.2, we showed that each node requires $O(\rho^{\log(c_r/c_T/c_r^2)})$ cost to find its relatives. As such, \textsc{CoverTreeToNetTree} runs in $O(k\rho^{\log(c_r/c_T/c_r^2)} n)$ time because $T'$ has $O(k\rho^{\log(c_r/c_T)} n)$ nodes and $c'_r \geq c_r$. Moreover, after the execution of \textsc{CoverTree-ToNetTree}, $T'$ has $O(k\rho^{\log(c'_r/c_T)} n)$ nodes. Therefore, \textsc{MakeSemiCompressed} runs in $O(k\rho^{\log(c'_r/c_T)} n)$ time. \hfill \Box
Chapter 4

Net-Tree Construction

In this chapter, we present algorithms to construct local net-trees. In Section 4.1, we show how an approximate Voronoi diagram can be built from a net-tree. In fact, the approximate Voronoi diagram associates each point to a node of a net-tree called its center. In Section 4.2, we propose an algorithm to construct a local net-tree in linear time ignoring the cost of finding centers also known as the point location. Then, we present two simple point location strategies which increase the time complexity of the construction algorithm to $O(n \log \Delta)$. In Section 4.3, we devise an approach to update the approximate Voronoi diagram after the insertion of a point to the tree and prove that the total expected cost of maintaining the approximate Voronoi diagram is $O(n \log n)$, which results in a randomized construction algorithm with the expected time complexity of $O(n \log n)$. Finally, we present our experimental results in Section 4.4 and compare the efficiency of our algorithms in practice.

Some results in this chapter are published in [49].
4.1 Approximate Voronoi Diagrams from Net-Trees

Many metric data structures naturally induce a partition of the search space. The use of hierarchies of partitions at different scales is a fundamental idea in the *approximate near neighbor* problem (also known as *point location in equal balls* (PLEB)) which is at the heart of many *approximate nearest neighbor* algorithms, including high dimensional approaches using locality-sensitive hashing [38, 40, 44, 45, 61].

Given a set of points $P$ and a query $q$, the *nearest neighbor* of $q$ in $P$ is the point $p \in P$ such that for all $p' \in P$, we have $d(q, p) \leq d(q, p')$. Relaxing this notion, $p$ is a $c$-approximate nearest neighbor (or $c$-ANN) of $q$ if for all $p' \in P$, we have $d(q, p) \leq cd(q, p')$.

The Voronoi diagram of a set of points $P$ is a decomposition of space into cells, one per point $p \in P$ containing all points for which $p$ is the nearest neighbor. The nearest neighbor search problem can be viewed as point location in a Voronoi diagram, though it is not necessary to represent the Voronoi diagram explicitly.

In this section we give a particular decomposition of space, an approximate Voronoi diagram from a net-tree. The purpose is not to introduce a new approximate Voronoi diagram (there are several already [3, 38, 61]), but rather to provide a clear description of the point location problem at the heart of our construction. Just as in Clarkson’s *sb* data structure [21], we will keep track of what “cell” contains each uninserted point. However, instead of using the Voronoi cells, we will use the approximate cells described below. Moreover, instead of having one cell per point, we have one cell per node, thus we can simulate having a Voronoi diagram of a net at each scale.

We want to associate points with the closest node in the tree that is close enough to be a relative. Ties are broken between nodes associated to the same point by always choosing the one that is lowest in the tree. Formally, we define the following function
mapping a point of the metric space $\mathcal{M}$ and a node to a pair of numbers.

$$f(x, p^\ell) := \begin{cases} (d(x, p), \ell) & \text{if } d(x, p) \leq c_r \tau^\ell \\ (+\infty, +\infty) & \text{otherwise} \end{cases}$$

The Voronoi cell of a node $p^\ell$ is then defined as

$$\text{Vor}(p^\ell) := \{ x \in \mathcal{M} | f(x, p^\ell) \leq f(x, q^m) \text{ for all } q^m \in T \},$$

where ordering on pairs is lexicographical. For a point $q \notin P$, the center for $q$ in $T$, denoted $C(q)$, is the node $p^\ell \in T$ such that $q \in \text{Vor}(p^\ell)$. As we will see in Section 4.2, finding the center of a point is the basic point location operation required to insert it into the net-tree. Fig. 4.1 illustrates the construction.

![Figure 4.1: The net-tree on the left induces the approximate Voronoi diagram on the right.](image)

The union of Voronoi cells $p^\ell$ for all $\ell$ gives an approximate Voronoi cell for the point $p$. The following lemma makes this precise.

**Lemma 4.1.1.** Let $T$ be a net-tree in $\text{LNT}(\tau, c_p, c_c)$ with $\tau \geq 2$, $c_c \geq c_p > 0$, and
\( c_r > \frac{c_r \tau}{\tau - 1} \) on a point set \( P \) in a metric space \( \mathcal{M} \). For any point \( q \in \mathcal{M} \), if \( C(q) = p^\ell \), then \( p \) is a \( (\frac{c_r \tau(\tau - 1)}{\tau - 1 - c_r \tau}) \)-ANN of \( q \) in \( P \).

**Proof.** Let \( m := \lceil \log_{c_r}(d(p, q)/c_r) \rceil \). Since \( C(q) = p^\ell \), \( d(p, q) \leq c_r \tau^\ell \) and as such, \( m \leq \ell \). Also, \( c_r \tau^{m-1} < d(p, q) \leq c_r \tau^{m} \). Recall \( N_m \) is the set of points in \( P \) associated with a node of level at least \( m \). Since \( p \in N_m \) and \( p^\ell \) is the center of \( q \), \( d(q, N_m) > c_r \tau^{m-1} \).

Furthermore, \( d(q, N_{m-1}) > c_r \tau^{m-1} \), because otherwise \( C(q) \) should be a node other than \( p^\ell \) so that the corresponding point belongs to \( N_{m-1} \), which contradicts the assumption. Also note that each node associated to a point in \( P \setminus N_{m-1} \) has an ancestor in a level at least \( m - 1 \). If the lowest ancestor in a level at least \( m - 1 \) is above \( m - 1 \), then it is the top of a jump, and the bottom node with the same associated point is in a level less than \( m - 1 \). Therefore, using Lemma 3.1.1, \( d_H(N_{m-1}, P) \leq c_r \tau^m / (\tau - 1) \). Now, using the triangle inequality,

\[
\begin{align*}
d(q, P) &\geq d(q, N_{m-1}) - d_H(N_{m-1}, P) \\
&> c_r \tau^{m-1} - \frac{c_r \tau^m}{\tau - 1} \quad [d(q, N_{m-1}) > c_r \tau^{m-1}] \\
&\geq \left( \frac{1}{\tau} - \frac{c_r \tau}{c_r (\tau - 1)} \right) d(p, q). \quad [d(p, q) \leq c_r \tau^m]
\end{align*}
\]

Therefore, \( d(p, q) < \frac{c_r \tau(\tau - 1)}{c_r (\tau - 1) - c_r \tau} d(q, P) \). \( \square \)

### 4.2 Bottom-up Insertion into a Net-Tree

Constructing a net-tree one point at a time has three phases. First, one finds the center (as defined in Section 4.1) of the new point. Second, the new point is inserted as a relative of its center, with its parent, children, and relatives computed by a constant-time local search. Third, new nodes associated with the point are added up the tree until the parent satisfies the covering property. In principle, this promotion phase can
propagate all the way to the root. Along the way, it is sometimes necessary to split a compressed edge to create a node that now has a relative (our new point) or remove an existing node that now has no relatives.

In the original work on net-trees, the difficult part of the algorithm finds not only the centers (or its equivalent), but also finds an ordering that avoids the propagation phase. Other algorithms have used the tree itself as the search structure to find the centers when needed [33], but this can lead to linear time insertions if the tree is deep.

We start this section by assuming the center of each new point is known and present the insertion and the propagation algorithms in Sections 4.2.1 and 4.2.2. Then, we will describe two simple point location algorithms in Section 4.3.

### 4.2.1 Insertion

In this section, we propose an algorithm to insert a new point \( q \) to a local net-tree. Once the center is found, \( q \) is added to the tree as indicated in Algorithm 6. In this algorithm, we first find the lowest level \( h \) in \( T \) that \( q \) has a relative (not itself) and the insertion of \( q \) at that level satisfies the packing property.

To ensure that the parent, children, and relatives of a new node are correct, \texttt{Update} procedure will be executed, see Fig. 4.2. In this procedure, we find relatives and children of a node \( p^\ell \) from \( \text{ch}(\text{rel}(\text{par}(p^\ell))) \) and \( \text{ch}(\text{rel}(p^\ell)) \), respectively. Also, the parent of \( p^\ell \) will be updated using Algorithm 2. Note that the second parameter of \texttt{FindParent} is \texttt{true} because we require the output to have a semi-compressed representation.

In the following lemma, we prove that \texttt{Update} procedure correctly works even if the last parameter is not the parent of \( p^\ell \). In fact, we only require a nearby node in a higher level as the last parameter. We use this procedure later in Section 4.2.2.

**Lemma 4.2.1.** Given a tree \( T \in \text{LNT}(\tau, c_p, c_c) \) with \( c_r \geq \frac{2c_c}{\tau-2} \). Let \( p^\ell \) and \( q^{\ell+1} \) be in \( T \) and \( d(p, q) \leq (c_c + c_r/\tau)\tau^{\ell+1} \). The \texttt{Update} procedure correctly finds the parent,
Figure 4.2: Finding the parent, relatives, and children of the new node $p^\ell$ with $q^\ell = C(p)$.
Algorithm 6 Insert a new point to a given net-tree

1: procedure Insert($T \in \text{LNT}(\tau, c_p, c_c, c_r, q)$)
2: \hspace{1em} $p^\ell \leftarrow C(q)$
3: \hspace{1em} $h \leftarrow \lceil \log_\tau (d(q, p)/c_r) \rceil$
4: \hspace{1em} if $d(p, q) \leq c_p \tau^h$ then
5: \hspace{2em} $h \leftarrow h - 1$
6: \hspace{1em} if $h < \ell - 1$ then
7: \hspace{2em} Create $p^{h+1}$ and add it as a child of $p^\ell$
8: \hspace{1em} if $h < \ell$ then
9: \hspace{2em} if $p^h \notin T$ then
10: \hspace{3em} Create $p^h$
11: \hspace{3em} Add $p^h$ as a child of $p^{h+1}$.
12: Create $q^h$ as a child of par($p^h$)
13: UPDATE($T, c_r, q^h, \text{par}(q^h)$)

Proof. First we prove that for all $x^\ell \in \text{rel}(p^\ell)$, we have $q^{\ell+1} \sim \text{par}(x^\ell)$, which means that the algorithm correctly finds the relatives of $p^\ell$. We have

$$d(q^{\ell+1}, \text{par}(x^\ell)) \leq d(q^{\ell+1}, p^\ell) + d(p^\ell, x^\ell) + d(x^\ell, \text{par}(x^\ell)) \quad \text{[triangle inequality]}$$

$$\leq (c_c + \frac{c_r}{\tau})\tau^{\ell+1} + c_r \tau^\ell + c_c \tau^{\ell+1} \quad \text{[relative & covering props.]}$$

$$= 2(c_c + \frac{c_r}{\tau})\tau^{\ell+1}$$

$$\leq c_r \tau^{\ell+1} \quad \text{[because } c_r \geq \frac{2c_c \tau}{\tau - 2}]$$

So $q^{\ell+1}$ and $\text{par}(x^\ell)$ are relatives.

Now, we show that if $x^{\ell+1}$ is the closest node to $p^\ell$ in level $\ell + 1$, then $x^{\ell+1} \sim q^{\ell+1}$, so the algorithm correctly finds the parent of $p^\ell$. We have $d(p, x) < d(p, q)$. By the
Algorithm 7 Update a net-tree after the insertion of a new node

1: procedure Update($T, c_r, p^\ell, q^{\ell+1}$)
2: for all $x^f \in \text{ch}(\text{rel}(q^{\ell+1}))$ do
3: if $d(p, x) \leq c_r \tau^\ell$ then
4: if $f < \ell$ then
5: Split the jump at $x^{\ell+1}$ at level $\ell$
6: Add $x^\ell$ to rel($p^\ell$) and $p^\ell$ to rel($x^\ell$)
7: for all $x^f \in \text{ch}(\text{rel}(p^\ell))$ do
8: $y^g \leftarrow \text{par}(x^f)$
9: if $d(p, x) < d(x, y)$ then
10: Change the parent of $x^f$ to $p^\ell$
11: if $|\text{ch}(y^g)| = 1$ and $|\text{ch}(x^f)| = 1$ and $|\text{rel}(x^f)| = 1$ then
12: Remove $x^f$ from the tree
13: FindParent($p^\ell, \text{true}$)

triangle inequality,

$$d(x, q) \leq d(x, p) + d(p, q)$$

$$< 2d(p, q) \quad \text{[parent property]}$$

$$\leq 2(c_c + \frac{c_r}{\tau})\tau^{\ell+1}$$

$$\leq c_r \tau^{\ell+1}, \quad \left[ c_r \geq \frac{2c_c \tau}{\tau - 2} \right]$$

which indicates that $x$ and $q$ are relatives at level $\ell + 1$.

Finally, we show that if $p^\ell$ is the closest node in level $\ell$ to a node $x^{\ell-1}$, then $p^\ell \sim \text{par}(x^{\ell-1})$, which implies that the algorithm correctly finds children of $p^\ell$. By the
parent property, \( d(p, x) < d(x^{\ell-1}, \text{par}(x^{\ell-1})) \). By the triangle inequality,

\[
\begin{align*}
\text{d}(p^\ell, \text{par}(x^{\ell-1})) & \leq \text{d}(p^\ell, x^{\ell-1}) + \text{d}(x^{\ell-1}, \text{par}(x^{\ell-1})) \\
& < 2\text{d}(x^{\ell-1}, \text{par}(x^{\ell-1})) & \text{[parent property]} \\
& \leq 2c_c \tau^\ell & \text{[covering property]} \\
& < c_r \tau^\ell.
\end{align*}
\]

Note that when we split a jump at level \( \ell \), we do not require to find relatives of this new node, because it was not present in the tree before the split operation and consequently, it does not have any relatives besides itself.

**Theorem 4.2.2.** Given a semi-compressed tree \( T \in \text{LNT}(\tau, c_p, c_c) \) with \( c_r \geq \frac{2c_c \tau}{\tau - 2} \) and an uninserted point \( q \) with \( p^\ell := C(q) \). Algorithm 6 inserts \( q \) into \( T \) and results a semi-compressed local net-tree \( T' \in \text{LNT}(\tau, c_p, c_c + \frac{c_r}{\tau}) \).

**Proof.** To prove this theorem, we need to show that the resulted tree satisfies the covering, the packing, and the parent invariants, also it is semi-compressed. Let \( x^{h+1} \) be the parent of \( q^h \) in \( T' \). By the parent property, \( d(q, x) < d(q^h, \text{par}(p^h)) \). Also,

\[
\begin{align*}
\text{d}(q^h, \text{par}(p^h)) & \leq \text{d}(q^h, p^h) + \text{d}(p^h, \text{par}(p^h)) & \text{[triangle inequality]} \\
& \leq c_r \tau^h + c_c \tau^{h+1} & \text{[relative and the covering properties]} \\
& = (c_c + c_r / \tau) \tau^{h+1}.
\end{align*}
\]

Therefore, \( d(q, x) < (c_c + c_r / \tau) \tau^{h+1} \), which implies that the covering constant in \( T' \) is \( (c_c + c_r / \tau) \). Note that the distance of every node \( p^\ell \in T' \) except \( q^h \) to its parent is at most \( c_c \tau^{\ell+1} \).

Let \( h \) be the minimum value so that \( d(q, p) \leq c_r \tau^h \). Then \( c_r \tau^{h-1} < d(p, q) \leq c_r \tau^h \), as such \( h = \lceil \log_c(d(p, q)/c_r) \rceil \). Insertion of \( q \) at level \( h \) should preserve the packing
property, i.e. \( d(p, q) > c_p \tau^h \). Since \( c_r \geq 2c_c \tau / (\tau - 2) \), we have \( c_p \tau^h < c_r \tau^h \). However, \( c_p \tau^h < c_r \tau^{h-1} \) does not necessarily hold, so if \( q \) is inserted at level \( h \), it may violate the packing property. Furthermore, we have

\[
d(p, q) > c_r \tau^{h-1} = \frac{1}{\tau} c_r \tau^h > \frac{1}{\tau} c_p \tau^h = c_p \tau^{h-1},
\]

which implies that insertion of \( q \) at level \( h - 1 \) satisfies the packing property.

To prove the parent property, we need to show that the parent of \( q^h \) in \( T' \) is correct and \( q^h \) cannot serve as the parent of any node with a level less than \( h - 1 \). Lemma 4.2.1 proves that the parent of \( q^h \) and its children are correct in \( T' \). Consider a node \( x^f \in T \) with \( y^{f+1} = \text{par}(x^f) \), where \( f \leq h - 2 \). We have \( d(p, y) > c_r \tau^{h-1} \), otherwise \( y^{f+1} \) should have been the center of \( q \). We have

\[
d(q, x) \geq d(q, y) - d(y, x) \quad \text{[triangle inequality]}
\]

\[
> \ c_r \tau^{h-1} - c_c \tau^{f+1} \quad \text{[covering property]}
\]

\[
\geq \frac{2c_c \tau}{\tau - 2} \tau^{h-1} - c_c \tau^{h-1} \quad \left[ \text{for } f \leq h - 2 \text{ and } c_r \geq \frac{2c_c \tau}{\tau - 2} \right]
\]

\[
= c_c \tau^{h-1}.
\]

Therefore, \( q \) cannot cover \( x^f \), which implies that we only need to check the nodes at level \( h - 1 \) to find children of \( q^h \).

Finally, it is easy to see that \( T' \) is semi-compressed, because the \textsc{Update} method removes those nodes that do not satisfy the semi-compressed condition.

4.2.2 The Bottom-up Propagation Algorithm

The insertion of a new point may violate the local covering property. When this happens, we promote the new node to higher levels of the tree. In this section, we
present a bottom-up propagation algorithm that handles the sequence of promotions. Here, there exists only one node at a time in the tree that violates the covering property, and we call it the violating node. As we proved in Theorem 4.2.2, after an insertion, the distance of a violating node $p^\ell$ to its parent is at most $(c_c + c_r/\tau)\tau^{\ell+1}$. Algorithm 8 describes the bottom-up propagation method. It receives a violating node $p^\ell$ as an argument and uses Algorithm 7 to find the parent, children, and relatives of a promoting node.

In this algorithm, iteration $i$ indicates promotion of point $p$ to level $\ell + i$, where $\ell$ is the level where $p$ was first inserted. In each iteration, we create a new node for $p$ in level $\ell + i$ and find its relatives, children, and parent. We set the parent of a violating node to the closest node in one level up. This assignment may turn the node $p^{\ell+i}$ into a violating node, and as such, we promote it to higher levels.

**Algorithm 8** Promote a violating node to restore the covering property

1: procedure BottomUpPropagation($T \in LNT(\tau, c_p, \frac{c_c \tau}{\tau - 1}), c_r, p^\ell$)
2:     for $i \leftarrow 1$ to $+\infty$ do
3:         $q^{\ell+i} \leftarrow \text{par}(p^{\ell+i-1})$
4:         if $d(p, q) \leq c_c\tau^{\ell+i}$ then
5:             break
6:     else
7:         Create node $p^{\ell+i}$
8:         $\text{par}(p^{\ell+i-1}) \leftarrow p^{\ell+i}$
9:         $x^h \leftarrow \text{par}(q^{\ell+i})$
10:        if $h > \ell + i + 1$ then
11:            Split the jump at $x^h$ at level $\ell + i + 1$
12:        UPDATE($T, c_r, p^{\ell+i}, \text{par}(q^{\ell+i})$)
Lemma 4.2.3. Given $c_r \geq c_c \tau / (\tau - 2)$, in the $i$-th iteration of Algorithm 8, we have

$$d(p^{\ell+i}, \text{par}(p^{\ell+i})) \leq (c_c + c_r / \tau) \tau^{\ell+i+1} \leq c_r \tau^{\ell+i+1}.$$ 

Proof. We use induction to prove this lemma. For the base case $i = 0$, Theorem 4.2.2 implies $d(p, \text{par}(p^\ell)) \leq (c_c + c_r / \tau) \tau^{\ell+1}$. Assume that the lemma holds for some $i - 1 \geq 0$, then we show that it is also true for $i$. In other words, the distance between $p^{\ell+i-1}$ to $q^{\ell+i} = \text{par}(p^{\ell+i-1})$ is greater than $c_c \tau^{\ell+i}$, as such $p$ should be promoted to level $\ell + i$. Note that the algorithm finds the parent of $p^{\ell+i}$ among relatives of $\text{par}(q^{\ell+i})$. Therefore, $\text{par}(p^{\ell+i})$ is a node at level $\ell + i + 1$ and $d(p^{\ell+i}, \text{par}(p^{\ell+i})) \leq d(p^{\ell+i}, \text{par}(q^{\ell+i}))$. So,

$$d(p^{\ell+i}, \text{par}(p^{\ell+i})) \leq d(p^{\ell+i}, \text{par}(q^{\ell+i})) \leq d(p^{\ell+i}, q^{\ell+i}) + d(q^{\ell+i}, \text{par}(q^{\ell+i})) \leq (c_c + c_r / \tau) \tau^{\ell+i+1} \leq c_r \tau^{\ell+i+1}.$$ 

Also, $c_c + \frac{c_r}{\tau} \leq c_r$ proves the lemma. \hfill \qed

The following lemma is a direct result of Lemma 4.2.3.

Lemma 4.2.4. In iteration $i$ of Algorithm 8, $p^{\ell+i} \sim q^{\ell+i}$.

In the following theorem, we prove the correctness of the bottom-up propagation algorithm.

Theorem 4.2.5. Given a semi-compressed tree $T \in \text{LNT}(\tau, c_p, c_c + \frac{c_r}{\tau})$ with $c_r \geq \frac{2c_c \tau}{\tau - 2}$. Let for all nodes $x^\ell \in T$ except $p^\ell$, $d(x^\ell, \text{par}(x^\ell)) \leq c_c \tau^{\ell+1}$ and for $p^\ell$, $d(p, \text{par}(p^\ell)) \leq (c_c + \frac{c_r}{\tau}) \tau^{\ell+1}$. Algorithm 8 turns $T$ into a semi-compressed tree $T' \in \text{LNT}(\tau, c_p, c_c)$. 50
Proof. The algorithm terminates, because the root at level $+\infty$ can cover any point. Also, it is clear from the algorithm that both the local packing and covering properties are satisfied in the output. Furthermore, Lemma 4.2.1 implies that the algorithm maintains the parent property and correctly finds relative. Note that in Algorithm 8, if the parent of $q^{\ell+i}$ is in a level greater than $\ell + i + 1$, we create a node for $q$ at level $\ell + i + 1$. In this case, we do not require to find relatives of this new node, because being semi-compressed implies that the new node $q^{\ell+i+1}$ does not have any relatives besides itself. Finally, we need to show that the resulted tree is semi-compressed. Lemma 4.2.4 indicates that all the create nodes for $p$ are required in the final semi-compressed representation. If the algorithm creates node $q^{\ell+i+1}$, then either it is the right parent and no promotion is needed or $p$ should be promoted to level $\ell + i + 1$. If promotion is required, then Lemma 4.2.4 implies that $p^{\ell+i+1}$ and $q^{\ell+i+1}$ are relatives. Therefore, the new node $q^{\ell+i+1}$ will be always in the output.

4.2.3 Analysis of the Construction

In this section, we prove that the running time of the construction algorithm ignoring point location is linear. As we will see later in this chapter, the point location step is the bottleneck of our net-tree construction algorithms.

Theorem 4.2.6. Not counting the PL step, the bottom-up construction algorithm runs in $O(\rho^{O(1)} n)$ time.

Proof. The running time depends on the number of iterations in the bottom-up propagation algorithm. Note that the algorithm may require $O(\log \Delta)$ iterations for each new point, because the number of levels in an uncompressed net-tree is $O(\log \Delta)$. A crude analysis suggests the running time $O(n \log \Delta)$ for the construction algorithm. However, we show that it only requires linear time.
In the following, we use an amortized analysis which imposes the cost of each iteration on a node in the output. In the promotion phase, Lemma 4.2.4 implies that every node of $p^{\ell+i}$ has at least one relative besides itself, namely $q^{\ell+i}$. So, we can make $q^{\ell+i}$ responsible to pay the cost of iteration $i$ for $p$. Note that a node $q^{\ell+i}$ will not be removed by any points that will be processed next, because $p^{\ell+i} \sim q^{\ell+i}$ satisfies the semi-compressed condition. In other words, there is always a node in the output that pays the cost of promotion. By Lemma 2.2.1, the cost of each iteration is $\rho^{O(1)}$ and $q^{\ell+i}$ has $\rho^{O(1)}$ relatives, as such $q^{\ell+i}$ receives $\rho^{O(1)}$ cost in total. Therefore, to pay the cost of all promotions for all $n$ points, each node in the output requires $\rho^{O(1)}$ charge. By Theorem 4.2.5, the output is semi-compressed and Theorem 3.1.3 implies that it has $O(\rho^{O(1)}n)$ size. Thus, the total cost of all promotions for all $n$ points does not exceed $O(\rho^{O(1)}n)$.

Notice that when a point is inserted to the tree for the first time, it does not necessarily have any other relatives. However, the insertion occurs only once for each point and it requires $\rho^{O(1)}$ time. Therefore, all insertions can be done in $O(\rho^{O(1)}n)$ time.

4.2.4 Point Location

In this section, we introduce two algorithms to find the center of an inserting point. These algorithms do not require a prior knowledge about all inserting points and the point location work is done only in the point location step, as such, we call them lazy point location methods. These lazy approaches dynamize our net-tree construction algorithm.

In the first algorithm, we find a path from the root of tree to the center. Our second algorithm is similar to [52] and starts from the root and creates a set of possible centers in each level. Both of these algorithms run in $O(\log \Delta)$ time for a single point.
Single Path

Here, we propose a point location algorithm that starts from the root and finds a path to the center so that the next node on the path is always in a lower level. In each step, we find a node in a lower level from the set of children of relatives of current node such that it can be the closest relative of $q$. Algorithm 9 describes this point location procedure.

Algorithm 9 Single path point location on a net-tree

1: procedure SinglePathPL($T \in \text{LNT}(\tau, c_p, c_c), c_r, q$)
2: $next \leftarrow current \leftarrow$ the root of $T$
3: $\ell \leftarrow$ level of $next$
4: while $d(q, next) \leq c_r \tau^\ell$ do
5: $current \leftarrow next$
6: $m \leftarrow \max_{p^\ell \in \text{ch(rel(current))}}\{\ell\}$
7: $next \leftarrow \arg\min_{p^\ell \in \text{ch(rel(current))}}\{d(q, p) | d(p, q) \leq c_r \tau^m\}$
8: return $current$

Theorem 4.2.7. Given $T \in \text{LNT}(\tau, c_p, c_c)$ with $c_r \geq \frac{2c_p \tau}{\tau - 2}$ and an uninserted point $q$. Algorithm 9 correctly finds $C(q)$ in $O(p^{O(1)} \log \Delta)$ time.

Proof. To prove this theorem, we first need to show that the next center of $q$ at some level $\ell$ can be found from the children of relatives of its current center at level $\ell+1$. Let $p^\ell+1$ and $t^\ell$ be the centers of $q$ at levels $\ell+1$ and $\ell$, respectively. Then, $d(p, q) \leq c_r \tau^{\ell+1}$
and $d(t, q) \leq c_r \tau^\ell$. Also, let $s^{\ell+1} := \text{par}(t^\ell)$. In the following, we prove that $p^{\ell+1} \sim s^{\ell+1}$.

\[
\begin{align*}
    d(s, p) &\leq d(s, q) + d(q, p) & \text{[triangle inequality]} \\
    &< 2d(s, q) & \text{[parent property]} \\
    &\leq 2(d(s, t) + d(t, q)) & \text{[triangle inequality]} \\
    &\leq 2(c_r \tau^{\ell+1} + c_r \tau^\ell) & \text{[covering property]} \\
    &\leq 2\left(\frac{c_r(\tau - 2)}{2\tau} + \frac{c_r}{\tau}\right)\tau^{\ell+1} & \text{[since $c_r \geq \frac{2c_c \tau}{\tau - 2}$]} \\
    &= c_r \tau^{\ell+1}.
\end{align*}
\]

Now, we prove that why we need the maximum level at Line 6 of the algorithm. Let $p^\ell := \text{current}$ be the top of a jump with the bottom node $p^h$. According to Fig 4.3, there may be another node at level $\ell - 1$ so that its the closest relatives of $q$. However, in the following we show that if $next = p^h$, or equivalently $p$ is the closest point to $q$ at level $\ell - 1$ with $d(p, q) \leq c_r \tau^{\ell-1}$, then the center of $q$ belongs to the subtree rooted at $p^\ell$ (i.e. $T_{p^\ell}$). In other words, we show for all $x^g \notin T_{p^\ell}$ with $g \leq \ell - 2$, $d(x, q) > c_r \tau^g$. Let $y^{\ell-1}$ be the ancestor of $x^g$ at level $\ell - 1$. We have $d(p, y) > c_r \tau^{\ell-1}$, otherwise $p$ and $y$ will be relatives at level $\ell - 1$ and $p^{\ell-1}$ should be present in the tree. We assume that $p$ is the closest point to $q$ at level $\ell - 1$, so $d(p, q) < d(y, q)$. We have

\[
\begin{align*}
    d(p, y) &\leq d(p, q) + d(y, q) & \text{[triangle inequality]} \\
    &< 2d(y, q) & \text{[parent property]} \\
\end{align*}
\]

, so

\[
    d(y, q) > \frac{d(p, y)}{2} > \frac{c_r \tau^{\ell-1}}{2}.
\]
We separate $g$ into two cases $g = \ell - 2$ and $g \leq \ell - 3$. For $g = \ell - 2$,

\[ d(q, x) \geq d(q, y) - d(y, x) \quad \text{[triangle inequality]} \]

\[ > \frac{c_r\tau^{\ell-1}}{2} - c_{cr}\tau^{\ell-1} \quad \text{[covering property]} \]

\[ \geq \left( \frac{c_r}{2} - \frac{c_r(\tau - 2)}{2\tau} \right)\tau^{\ell-1} \]

\[ = c_r\tau^{\ell-2}. \]

So, $q$ cannot be a relative of $x^g$.

For $g \leq \ell - 3$,

\[ d(q, x) \geq d(q, y) - d(y, x) \quad \text{[triangle inequality]} \]

\[ > \frac{c_r\tau^{\ell-1}}{2} - \frac{c_{cr}\tau^{\ell-1}}{\tau - 1} \quad \text{[Lemma 3.1.1]} \]

\[ \geq \left( \frac{c_r}{2} - \frac{c_r(\tau - 2)}{2\tau(\tau - 1)} \right)\tau^{\ell-1} \]

\[ = \frac{c_r}{2(\tau - 1)}\tau^{\ell-1} \]

\[ > c_r\tau^{\ell-3}. \]

Therefore, again $q$ cannot be a relative of $x^g$.

By Lemma 2.2.1, the size of children and relatives for each node is $\rho^{O(1)}$. Since a net-tree may have $O(\log \Delta)$ levels, this point location algorithm runs in $O(\rho^{O(1)} \log \Delta)$ time.

\[ \square \]

**Multipath**

This algorithm starts from the root and follows multiple paths in the net-tree to find the center of a given point. In this point location method, we only use the children information to build a set of possible centers at each level. Algorithm 10 describes this
Figure 4.3: The approximate Voronoi diagram on the right corresponds to the net-tree on the left. In this figure, \( q \) is a point not in \( T \) and \( p^\ell \sim s^\ell \). (a) A part of a net-tree. (b) The partial approximate Voronoi diagram. The cells at level \( \ell \) and \( \ell - 1 \) are illustrated using solid and dashed lines, respectively. At level \( \ell \), \( q \) belongs to the cell of \( p^\ell \), but at level \( \ell - 1 \), it is in the cell of \( t^{\ell-1} \).

method. We invoke this algorithm for the first time using

\[ \text{MULTIPATHPL}(T, c_r, q, \{\text{the root of } T\}, +\infty). \]

In this procedure, the lines 7 to 10 ensure if \( p^\ell \) is the top of a jump and there is a node in \( \text{ch}(Q) \) with a level greater than \( \ell \), then instead of \( p^\ell \), we add \( \text{par}(p^\ell) \) to the set of all possible centers at the next level because \( p^m \notin T \).

**Theorem 4.2.8.** Let \( T \in \text{LNT}(\tau, c_p, c_c) \) with \( c_r \geq \frac{2c_c\tau}{\tau - 2} \) and \( q \) be a point not in \( T \). Algorithm 10 correctly finds \( C(q) \) in \( O(\rho^{O(1)} \log \Delta) \) time.

**Proof.** Let \( x^h := C(q) \). Then, at least one ancestor of \( x^h \) is in the set of possible centers
### Algorithm 10 Multipath point location on a net-tree

1: procedure \textsc{MultipathPL}(T \in \mathbf{LNT}(\tau, c_p, c_c, c_r, q, Q, \ell))

2: \hspace{1em} if $Q \neq \emptyset$ and $d(q, Q) \leq c_r \tau^\ell$ then

3: \hspace{2em} $m \leftarrow \max_{p^\ell \in \text{ch}(Q)} \{\ell\}$

4: \hspace{2em} $Q' \leftarrow \emptyset$

5: \hspace{2em} for $p^\ell \in \text{ch}(Q)$ do

6: \hspace{3em} if $d(p, q) \leq c_r \tau^m$ then

7: \hspace{4em} if $\ell = m$ then

8: \hspace{5em} $Q' \leftarrow Q' \cup \{p^\ell\}$

9: \hspace{4em} else

10: \hspace{5em} $Q' \leftarrow Q' \cup \{\text{par}(p^\ell)\}$

11: \hspace{2em} $\text{center} \leftarrow \textsc{MultipathPL}(T, c_r, q, Q', m)$

12: \hspace{2em} if $\text{center} = \text{none}$ then

13: \hspace{3em} $\text{center} \leftarrow \arg\min_{p^\ell \in Q} d(q, Q)$

14: \hspace{2em} return $\text{center}$

15: return none

---

$Q$ at some level $\ell$. Note that such ancestor can be the root of $T$. If $h = \ell - 1$, then

\[
\begin{align*}
    d(\text{par}(x^{\ell-1}), q) &\leq d(\text{par}(x^{\ell-1}), x^{\ell-1}) + d(x, q) & \text{[triangle inequality]} \\
    &\leq c_c \tau^\ell + c_r \tau^{\ell-1} & \text{[covering property]} \\
    &< c_r \tau^\ell. & \left[ c_r \geq \frac{2c_c \tau}{\tau - 2} \right]
\end{align*}
\]

Therefore, the algorithm will correctly find $x^{\ell-1}$ as the center of $q$. For $h \leq \ell - 2$, suppose that $y^{\ell-1}$ is the ancestor of $x^h$ at level $\ell - 1$. According to this theorem, $y^{\ell-1}$ should be in the set of next possible centers. For contradiction, assume that $y^{\ell-1}$ is
not in the next $Q$, i.e. $d(q,y) > c_r\tau^\ell - 1$. We have

\[
d(q,x) \geq d(q,y) - d(y,x) \geq c_r\tau^\ell - 1 - \frac{c_r\tau(\tau - 2)}{2(\tau - 1)}\tau^\ell - 1 = \frac{c_r\tau}{2(\tau - 1)}\tau^\ell - 1 > c_r\tau^\ell - 2.
\]

It contradicts with our assumption since $d(q,x) \leq c_r\tau^h \leq c_r\tau^\ell - 2$. Therefore, the algorithm correctly finds the center of $q$. The time complexity follows from the number of levels and the size of children in $T$.

\[\square\]

4.3 Randomized Incremental Construction

In this section, we show how to eagerly compute the centers of all uninserted points. In fact, we present an approach to update the approximate Voronoi diagram with respect to uninserted points after the insertion of a new point to the net-tree. The centers are updated each time either a new node is added or an existing node is deleted by doing a local search among parents, children, and relatives of the node. We show that the following invariant is satisfied after each insertion or deletion.

**Invariant.** The centers of all uninserted points are correctly maintained.

In Section 4.3.1, we present the point location algorithm. Then, in Section 4.3.2, we show that for a random ordering of points, the point location takes $O(n \log n)$ time in expectation. As this point location work is the main bottleneck in the algorithm, the following theorem summarizes the result of the randomized incremental approach.
Theorem 4.3.1. Given a random permutation $\pi = \langle p_1, \ldots, p_n \rangle$ from a point set $P$ in a doubling metric space $\mathcal{M}$. A net-tree $T \in \text{NT}(\tau, \frac{c_p(\tau-1)-2c_r}{2(\tau-1)}, \frac{c_r\tau}{\tau-1})$ with $c_r = \frac{2c_p\tau}{\tau-4}$ can be constructed from $\pi$ in $O(p^{O(1)}n \log n)$ expected time, where $\tau \geq \max\{5, \frac{2c_p}{c_p} + 2\}$ and $0 < c_p \leq c_c < \frac{c_p(\tau-1)}{2}$ are constants.

4.3.1 The Point Location Algorithm

We will describe a simple eager point location algorithm in this section. In this algorithm, we maintain the approximate Voronoi diagram with respect to uninserted points and whenever a node is added to or deleted from tree, we update the diagram accordingly. More specifically, we store the center of each uninserted point, and for each node, a list of uninserted points whose center is that node (i.e. the Voronoi cell of the node). The cell of a node $p^\ell$, denoted $S(p^\ell, T)$, is the list of points in $\text{Vor}(p^\ell)$. We partition the points $x$ of $S(p^\ell, T)$ into $S_{in}(p^\ell, T)$ and $S_{out}(p^\ell, T)$ depending on whether $d(p, x) \leq c_p \tau^{\ell-1}/2$ or not. In other words,

$$S_{in}(p^\ell, T) := \{ q \in S(p^\ell, T) \mid 0 < d(p, q) \leq \frac{c_p \tau^{\ell-1}}{2} \},$$

$$S_{out}(p^\ell, T) := \{ q \in S(p^\ell, T) \mid \frac{c_p \tau^{\ell-1}}{2} < d(p, q) \leq c_r \tau^{\ell} \}.$$

This separation saves some unnecessary distance computations.

Each time a node is added to the tree $T$ to create a new tree $T'$, we update the centers and cells nearby. There are two different ways that a new node is created, either it splits a jump or it is inserted as a child of an existing node, see Fig. 4.4. If a jump from $p^h$ to $p^\rho$ is split at level $\ell$, then we select $S(p^\ell, T')$ from the nodes of $S(p^h, T)$. If $p^\ell$ is inserted as a child of $s^{\ell+1}$, then

$$S(p^\ell, T') \subset \left\{ \bigcup S_{out}(x^h, T) \mid x^h \in \text{rel}(s^{\ell+1}) \cup \text{ch}(\text{rel}(s^{\ell+1})) \cup \text{ch}(\text{rel}(p^\ell)) \right\}.$$
A node \( p^\ell \) with parent \( p^h \) may be removed if required by the compression. In such cases, \( S_{in}(p^\ell, T) \) is added to \( S_{in}(p^h, T') \) and the points in \( S_{out}(p^\ell, T) \) are tested to determine which points belong to \( S_{in}(p^h, T') \) or \( S_{out}(p^h, T') \), see Fig 4.4.

![Diagram showing different cases in the eager point location algorithm.](image)

Figure 4.4: Different cases in the eager point location algorithm. White dots are uninserted points.

The following lemma shows that the point location algorithm correctly maintains the invariant.

**Lemma 4.3.2.** The point location algorithm correctly maintains the invariant after
the insertion or deletion of a node.

Proof. First, we prove that after the deletion of a node $p^\ell$, the point location algorithm correctly updates the center of each uninserted point $q \in S(p^\ell, T)$. Let $p^h$ be the parent of $p^\ell$. Note that by the definition of a center, $p^h$ should be the new center for $q$ after the deletion of $p^\ell$. If $q \in S_{in}(p^\ell, T)$, then $d(p, q) \leq c_p \tau^{\ell-1}/2 < c_p \tau^{h-1}/2$, which means $q \in S_{in}(p^h, T')$. Otherwise, $q$ can belong to the inner or the outer cell of $p^h$.

Now, we prove that if $p^\ell$ is added and $q \in S(p^\ell, T')$, then $q$ belongs to the set of nearby uninserted points of $p^\ell$.

(a) $p^\ell$ splits a jump from $p^h$ to $p^g$: Since $q \in S(p^\ell, T')$ and $T'$ has only one node $p^\ell$ more than $T$, $q$ should have been in a cell of a node of $p$. Also, $d(p, q) > c_r \tau^g$, because otherwise $p^g$ should be the center of $q$ in $T$. So, $d(p, q) \leq c_r \tau^\ell < c_r \tau^h$, which means $q \in S(p^h, T)$.

(b) $p^\ell$ is inserted as a child of $s^\ell+1$: First we show that $d(s, q) \leq c_r \tau^{\ell+1}$. We have

$$d(s, q) \leq d(s, p) + d(p, q) \quad \text{[triangle inequality]}$$

$$\leq \left(c_c + \frac{c_r}{\tau}\right) \tau^{\ell+1} + c_r \tau^\ell \quad \text{[} q \in S(p^\ell, T) \text{ and Lemma 4.2.3]}$$

$$= \left(c_c + 2\frac{c_r}{\tau}\right) \tau^{\ell+1}$$

$$\leq c_r \tau^{\ell+1} \quad \text{[for } c_r \geq \frac{c_c \tau}{\tau - 2}, c_c + \frac{2c_r}{\tau} \leq c_r \text{]}.$$

So, there exists at least one node in level $\ell + 1$ that can be served as the center of $q$ before the insertion of $p^\ell$, and it is $s^{\ell+1}$. However, $q$ might be closer to any other nodes, so $q$ is not necessarily in $S(s^{\ell+1}, T)$.

If $q \in S(x^h, T)$, then we show that $\ell - 1 \leq h \leq \ell + 1$. Suppose for contradiction,
\( h < \ell - 1 \). Then,

\[
\begin{align*}
  d(p, x) &\leq d(p, q) + d(q, x) & \text{[triangle inequality]} \\
  &< d(q, x) + d(q, x) & [d(p, q) < d(q, x) \text{ because } q \in \mathcal{S}(p^\ell, T')] \\
  &\leq 2c_r \tau^h & [q \in \mathcal{S}(x^h, T)] \\
  &\leq c_r \tau^{\ell-1} & [h \leq \ell - 2 \text{ and } \tau \geq 2]
\end{align*}
\]

Therefore, \( p^{\ell-1} \sim x^{\ell-1} \) and \( q \in \mathcal{S}(p^{\ell-1}, T) \), which is a contradiction because \( p \) is inserted at level \( \ell \).

Suppose for contradiction, \( h > \ell + 1 \). Then,

\[
\begin{align*}
  d(x, s) &\leq d(x, q) + d(q, s) & \text{[triangle inequality]} \\
  &< d(q, s) + d(q, s) & [d(x, q) < d(q, s) \text{ because } q \in \mathcal{S}(x^h, T)] \\
  &\leq 2(d(q, p) + d(p, s)) & \text{[triangle inequality]} \\
  &\leq 2\left(\frac{c_r}{\tau} + c_c + \frac{c_r}{\tau}\right) \tau^{\ell+1} & \text{[by Lemma 4.2.3, } d(p, s) \leq (c_c + \frac{c_r}{\tau}) \tau^{\ell+1}] \\
  &\leq c_r \tau^{\ell+1}. & \text{[for } c_r \geq \frac{2c_c \tau}{\tau - 4}, \frac{2c_r}{\tau} \leq c_r]\end{align*}
\]

So, \( s^{\ell+1} \sim x^{\ell+1} \). Also, \( d(q, x) < d(q, s) \leq c_r \tau^{\ell+1} \). Therefore, \( q \in \mathcal{S}(x^{\ell+1}, T) \), which is a contradiction. It is easy to see that \( x^h \) belongs to

\[
\text{rel}(s^{\ell+1}) \cup \text{ch}(\text{rel}(s^{\ell+1})) \cup \text{ch}(\text{rel}(p^\ell)).
\]

Finally, we prove that the points in cell \( \mathcal{S}(p^\ell, T') \) are in the outer cells of the nearby nodes. For contradiction, suppose that \( q \in \mathcal{S}_m(x^h, T) \), where \( \ell - 1 \leq h \leq \ell + 1 \). So, \( d(q, x) \leq c_p \tau^{h-1}/2 \). Then, \( d(p, q) < d(q, x) \) and by the triangle
inequality,
\[ d(p, x) \leq d(p, q) + d(q, x) < 2d(q, x) \leq c_p \tau^{h-1}. \]

If \( \ell \leq h \leq \ell + 1 \), then \( d(p, x) \leq c_p \tau^\ell \) and it contradicts with the packing property at level \( \ell \). Otherwise, if \( h = \ell - 1 \), then \( d(p, x) \leq c_p \tau^{\ell-2} \) and it also contradicts with the packing property at level \( \ell - 1 \). \( \square \)

### 4.3.2 Analysis of the Point Location Algorithm

When a node of \( p \) checks an uninserted point \( q \) to see if \( q \) belongs to its cell, we say \( p \) touches \( q \). To analyze the point location algorithm, we should count the total number of touches, because each touch corresponds to a distance computation. Note that a point does not change its center each time it is touched. This is the main challenge in the point location, to avoid touching a point too many times unnecessarily.

We classify the touches into three groups of basic touches, split touches, and merge touches. Then, we use a backwards analysis to bound the expected number of such touches. The standard approach of using backwards analysis for randomized incremental constructions will not work directly for the tree construction, because the structure of the tree is highly dependent on the order the points were added. Instead, we define random events that can happen for each point \( p_i \) of a permutation \( \langle p_1, \ldots, p_n \rangle \) in \( P_j := \{ p_1, \ldots, p_j \} \), where \( j < i \). These events are defined only in terms of the points in the permutation, and do not depend on a specific tree. We show that each point is involved in \( O(\log n) \) such events. Later, we show that the touches all correspond to these random events.

If \( p_i \) is touched by a new point \( p_j \), then we say a basic touch has happened, see Fig. 4.5a. If \( p_i \) is touched by the point of \( C(p_i) \) after the insertion of \( p_j \), then a split touch has happened, see Fig. 4.5b. Intuitively, a split touch in the tree occurs when
Figure 4.5: The approximate Voronoi diagrams at the top are induced on the part of net-trees at the bottom. White dots show the uninserted points. (a) The insertion of \( p_j \) at level \( \ell \) results a basic touch from \( p_j \) on \( p_i \). Before the insertion, \( p_i \) and \( p_j \) belong to cell \( S(p_\ell^{k+1}, T) \), and after that, \( p_i \) remains in the same cell. (b) The insertion of \( p_j \) at level \( \ell \) results a split touch from \( p_k \) on \( p_i \). Before the insertion, \( p_i \) and \( p_j \) belong to cell \( S(p_\ell^k, T) \), and after that, \( p_i \) remains in the same cell.
$C(p_i)$ is the top of a jump and the insertion of $p_j$ results that jump to be split at a lower level. By the point location algorithm, the cell of a new node can be found from the cell of its parent. Therefore, $p_i$ and all other points in the cell of $C(p_i)$ will be touched by the point of $C(p_i)$ at a smaller scale. A split touch is either below or above, which will be discussed later. Similarly, If $p_i$ is touched by the point of $C(p_i)$ after the deletion of $C(p_i)$ triggered by the insertion of $p_j$, then a merge touch has happened. In other words, a merge touch occurs if the insertion of $p_j$ results $C(p_i)$ to be deleted and its adjacent edges merged to a jump. In this case, the point location algorithm moves $p_i$ and other points in the cell of $C(p_i)$ to the cell of the parent of $C(p_i)$. For the sake of simplicity, we abuse the notion of touches for split and merge cases in the following way. If in a split or merge touch, the point of $C(p_i)$ touches $p_i$, then we charge $p_j$ for that touch and we say that $p_j$ touches $p_i$ (since $p_j$ has triggered that touch).

**Lemma 4.3.3.** A point $p_j$ can touch $p_i$ at most $\rho^{O(1)}$ times.

**Proof.** First we count the number of basic touches. Note that when we promote $p_j$ to a higher level, $p_j$ might touch $p_i$ more than once. From the algorithm in Section 4.3.1, the promoting node only checks nearby cells from one level down to one level up. Therefore, $p_j$ can only touch $p_i$ at most three times.

Now, we compute the number of split touches. When $p_j$ splits a jump on $C(p_i)$, it may create two new nodes for $C(p_i)$, see Fig. 4.5b. So, $p_i$ can be touched by $p_j$ at most twice. If $p_j$ requires to be promoted to a higher level, $p_i$ may receive more touches. This case only happens when $p_i$ is touched, but its center remains unchanged. Let $p_j$ be inserted at level $\ell$ and $q^{\ell+1} := \text{par}(p_j^{\ell})$. From Lemma 4.2.3, $d(p_j, q) \leq c_r \tau^\ell + c_c \tau^{\ell+1}$.

In the following, we will show that in the promotion process, $q$ cannot touch $p_i$ more than $\log_c(c_r/c_c + \tau)$ times. To prove this bound, we show that the promotion cannot continue more than $k > 1$ levels above $\ell$ with the same parent $q$. In other words, $d(p_j, q) \leq c_c \tau^{\ell+k}$, which satisfies the covering property. So $c_r \tau^\ell + c_c \tau^{\ell+1} \leq c_c \tau^{\ell+k}$,
which results $k \geq \lceil \log_\tau(c_r/c_c + \tau) \rceil$. Therefore, the total number of split touches from $p_j$ on $p_i$ is also constant.

Finally, we prove that the number of merge touches is also constant. Recall that when a node is deleted from the tree, the point location algorithm only checks the uninserted points in its outer cell to determine which points should be moved to the inner or outer cells of its parent. Let $p^\ell_m$ be the node to be deleted and $p_i \in S(p^\ell_m, T)$. Here, we wish to find a level $\ell + k$, where $k > 0$, such that $p_i$ goes to the inner cell of $p_{m}^{\ell+k}$ and as such does not receive more merge touches from $p_j$. Therefore, $d(p_i, p_m) \leq c_p \tau^{\ell+k-1}/2$. By the definition of a center, $p_i$ cannot be in a distance farther that $c_r \tau^\ell$ from $p_m$. So, we have $c_r \tau^\ell \leq c_p \tau^{\ell+k-1}/2$, which results $k \geq 1 + \log_\tau(2c_r/c_p)$. Therefore, $p_j$ can only touch $p_i$ a constant number of times via merge touches. 

In this section, our goal is finding the expected number of touches for each uninserted point in a random permutation. The following theorem summarizes the required cost of the point location step.

**Theorem 4.3.4.** The expected running time of point location in the randomized incremental construction algorithm is $O(\rho^{O(1)}n \log n)$.

**Basic Touches**

In this section, we first prove that the distance of every point touching $p_i$ with a basic touch is bounded by the distance of $p_i$ to its nearest neighbor among the inserted points. Using this observation, we divide a permutation into phases, where each phase is an interval in which the nearest neighbor of $p_i$ remains unchanged, see Fig. 4.6. We show that the number of basic touches on $p_i$ in each phase is constant. Then, using a backwards analysis we show that the expected number of phases for each point in $O(\log n)$. Therefore, the expected number of basic touches for all points is $O(n \log n)$. 

66
Figure 4.6: The solid and white dots show the inserted and uninserted points, respectively. An arrow shows the nearest neighbor of $p_i$ among the inserted points. The uninserted points in each ball are the only points that may touch $p_i$ via a basic touch. With the change of the nearest neighbor, the ball containing touching points shrinks. The points that get inserted from the left figure to the right figure belong to the same phase.

**Lemma 4.3.5.** If $p_j$ touches $p_i$ via a basic touch, then $d(p_i, p_j) < \frac{24c_r r^4 (\tau - 1)}{c_p (\tau + 2) (\tau - 4)} d(p_i, P_{j-1})$

**Proof.** Let $p_k^\ell$ be the center of $p_i$ in $P_{j-1}$, where $k < j < i$. According to the PL algorithm, $p_j$ can be in any level between $\ell - 1$ and $\ell + 1$. If $p_j$ is at level $\ell + 1$, then $p_k \in \text{ch}(\text{rel}(p_j^{\ell+1}))$. So,

$$d(p_j, p_k) \leq d(p_j^{\ell+1}, \text{par}(p_k^\ell)) + d(\text{par}(p_k^\ell), p_k^\ell) \quad \text{[triangle inequality]}$$

$$\leq c_r \tau^{\ell+1} + c_c r^{\ell+1} \quad \text{[relative and covering properties]}$$

$$< 2c_r \tau^{\ell+1}.$$
If $p_j$ is at level $\ell - 1$, then $p_k \in \text{rel(par}(p_{j}^{\ell-1})))$. We have

$$d(p_j, p_k) \leq d(p_j^{\ell-1}, \text{par}(p_j^{\ell-1})) + d(\text{par}(p_j^{\ell-1}), p_k^{\ell}) \quad \text{[triangle inequality]}$$

$$\leq (c_c + \frac{c_r}{\tau})\tau^\ell + c_r\tau^\ell \quad \text{[relative property and Lemma 4.2.3]}$$

$$< 2c_r\tau^\ell. \quad \text{[for } c_r \geq \frac{c_c\tau}{\tau - 1}, c_c + \frac{c_r}{\tau} \leq c_r]\$$

If $p_j$ is at level $\ell$, then

$$d(p_j, p_k) \leq d(p_j^\ell, \text{par}(p_j^\ell)) + d(\text{par}(p_j^\ell), \text{par}(p_k^\ell)) + d(\text{par}(p_k^\ell), p_k^\ell) \quad \text{[triangle inequality]}$$

$$\leq (c_c + \frac{c_r}{\tau})\tau^{\ell+1} + c_r\tau^{\ell+1} + c_c\tau^{\ell+1} \quad \text{[Lemma 4.2.3]}$$

$$< 2c_r\tau^{\ell+1}. \quad \text{[} c_r = \frac{2c_c\tau}{\tau - 4}]$$

Therefore, we have $d(p_j, p_k) < 2c_r\tau^{\ell+1}$ for all cases, as such

$$d(p_i, p_j) \leq d(p_i, p_k) + d(p_k, p_j) \leq c_r\tau^\ell + 2c_r\tau^{\ell+1} < 3c_r\tau^{\ell+1}.$$

As we saw earlier in Section 4.3.1, $p_j$ touches $p_i$ with a basic touch if $p_i$ is in the outer cell of $p_k$, which means $d(p_i, p_k) > c_p\tau^{\ell-1}/2$. Combining the last two inequalities results $d(p_i, p_j) < 6c_r\tau^2 d(p_i, p_k)/c_p$. From Lemma 4.1.1, for $c_r = \frac{2c_c\tau}{\tau - 4}$, we have $d(p_i, p_k) < 2\tau(\tau - 1)/(\tau + 2)d(p_i, P_{j-1})$. The lemma follows from the last two inequalities.

\[\Box\]

**Lemma 4.3.6.** If $d(p_i, P_k) = d(p_i, P_j)$ for $k < j < i$, then the number of basic touches on $p_i$ from $p_k$ to $p_j$ is $\rho^{O(1)}$.

**Proof.** Let $q$ be the closest point to $p_i$ in both $P_k$ and $P_j$. Also, we define $h := \lceil \log_\tau(d(p_i, q)/c_r) \rceil$. Note that the center of $p_i$ in $P_j$ cannot be in a level lower than $h$ because otherwise, $q$ will not be the closest point to $p_i$ anymore. If two points
x, y ∈ \{p_k, \ldots, p_j\} touch p_i at levels f and g and the minimum distance remains unchanged, then f, g ≥ h − 1 because the point location algorithm in Section 4.3.1 checks the nearby cells from one level down to one level up. By the packing property, d(x, y) > c_pτ^{h-1}. By definition, d(q, p_i) ≤ c_τ h. The last two inequalities result d(x, y) > \frac{c_p}{c_τ} d(q, p_i). By Lemma 4.3.5, every touching point in \{p_k, \ldots, p_j\} is within distance \frac{24\rho c^4(τ−1)}{c_p(τ+2)(τ−4)}d(p_i, P_j) from p_i. Using the last two inequalities and the Packing Lemma, there are a constant number of points from p_k to p_j that can touch p_i but not changing the minimum distance.

**Theorem 4.3.7.** The total expected number of basic touches in a random permutation is \(O(\rho^{O(1)}n \log n)\).

**Proof.** Using Lemma 4.3.6, only a constant number of basic touches on a point p_i can occur before the distance from p_i to the inserted points must go down. Therefore, it suffices to bound \(E[|\{j \mid d(p_i, P_{j-1}) \neq d(p_i, P_j)\}|]\). We observe that \(d(p_i, P_{j-1}) \neq d(p_i, P_j)\) only if \(p_j\) is the unique nearest neighbor of \(p_i\) in \(P_j\). Using a standard backwards analysis, this event occurs with probability 1/j. Therefore,

\[
E[|\{j \mid d(p_i, P_{j-1}) \neq d(p_i, P_j)\}|] \leq \sum_{j=1}^{n} \frac{1}{j} = O(\log n).
\]

By Lemma 4.3.3, each point \(p_j\) may cause a constant number of basic touches on \(p_i\). So, the expected number of basic touches on \(p_i\) is \(O(\rho^{O(1)} \log n)\), which results \(O(\rho^{O(1)}n \log n)\) touches in expectation for the permutation.

**Split and Merge Touches**

In this section, we define bunches near a point \(p_i\). These bunches are sufficiently-separated disjoint groups of points around \(p_i\). We show that each point has a constant number of bunches nearby. Then, we define two random events for each point based
Figure 4.7: The solid dots are points of $P_j$ and white dots are points in $P \setminus P_j$. $B$ and $B'$ are two bunches near $p_i$ centered at $x$ and $x'$. The smaller balls contain the points in the corresponding bunches. No points of $P_j$ lies in the shaded region.

on its nearby bunches. We prove that the expected number of events for each point of a permutation is $O(\log n)$. Then, we show that the number of split touches can be counted by such events. Finally, we prove that the number of merge touches can be bounded in terms of the number of split touches.

**Definition 4.3.8.** $B \subseteq P_j$ is a bunch near $p_i$, if there exists a center $x \in B$ such that

1. $B = B(x, \alpha d(p_i, x)) \cap P_j$,
2. $[B(x, \beta d(p_i, x)) \setminus B(x, \alpha d(p_i, x))] \cap P_j = \emptyset$,
3. $d(p_i, x) \leq \frac{2\tau(\tau-1)}{\tau+1}d(p_i, P_j)$,

where $j < i$, $0 < \alpha \leq 0.5$ and $\beta \geq 2\alpha$.

See Fig. 4.7 for an illustration of bunches. Note that the third property is the result of Lemma 4.1.1 and $c_r = \frac{2\alpha \tau}{\tau-1}$. Next, we show that there is a constant number of bunches near each point in a permutation.

**Lemma 4.3.9.** Let $B$ and $B'$ be two distinct bunches with centers $x$ and $x'$ near $p_i$ in $P_j$, respectively. Then $d(x, x') > \frac{\alpha}{\alpha+1} \max\{d(p_i, x), d(p_i, x')\}$.
Proof. Without loss of generality, let $d(p_i, x) > d(p_i, x')$. Suppose for contradiction,

$$d(x, x') \leq \frac{\alpha}{\alpha + 1} d(p_i, x) < d(p_i, x).$$

The first property of a bunch results $x' \in B$. Let $y' \in B' \setminus B$. The second property of $B$ results $d(x, y') > d(p_i, x)$. Using the triangle inequality,

$$d(x', y') \geq d(y', x) - d(x', x) > d(p_i, x) - \frac{\alpha}{\alpha + 1} d(p_i, x) = (\beta - \frac{\alpha}{\alpha + 1}) d(p_i, x). \quad (4.1)$$

Also, using the first property of $B'$ and the triangle inequality,

$$d(x', y') \leq d(p_i, x') \leq d(p_i, x) + d(x, x') \leq (1 + \frac{\alpha}{\alpha + 1}) d(p_i, x) \quad (4.2)$$

By (4.1) and (4.2),

$$(\beta - \frac{\alpha}{\alpha + 1}) d(p_i, x) < (1 + \frac{\alpha}{\alpha + 1}) d(p_i, x).$$

Therefore, $\beta < \frac{\alpha}{\alpha + 1} + \frac{2\alpha^2 + \alpha}{\alpha + 1} = 2\alpha$, which is a contradiction because by definition $\beta \geq 2\alpha$. Thus, $d(x, x') > \frac{\alpha}{\alpha + 1} d(p_i, x)$, as required. \qed

Lemma 4.3.10. For some constants $\alpha$ and $\beta$, there are $\rho^{O(1)}$ bunches near $p_i$.

Proof. From Lemma 4.3.9, for any bunches $B$ and $B'$ near $p_i$ with centers $x$ and $x'$, we have

$$d(x, x') > \frac{\alpha}{\alpha + 1} \max\{d(p_i, x), d(p_i, x')\} \geq \frac{\alpha}{\alpha + 1} d(p_i, P_j).$$

So, the third property of bunches and the Packing Lemma imply the bound. \qed

The following lemma shows that each jump in a local net-tree corresponds to an empty annulus around the corresponding point. We will use this lemma to show the
relation between jumps in a net-tree and bunches in a point set.

Lemma 4.3.11. In a semi-compressed local net-tree $T \in \text{LNT}(\tau, c_p, c_c)$ with $c_r = \frac{2c_c\tau}{\tau - 4}$, if there is a jump from $p^\ell$ to $p^h$, then

$$P_{p^h} \subset B(p, \frac{1}{2}c_r\tau^h)$$

and

$$B(p, \frac{1}{2}c_r\tau^{\ell-1}) \setminus B(p, \frac{1}{2}c_r\tau^h) = \emptyset.$$ 

Proof. Using Lemma 3.1.1,

$$P_{p^h} \subseteq B(p, \frac{c_c\tau}{\tau - 1} \tau^h) = B(p, \frac{c_r(\tau - 4)}{2(\tau - 1)} \tau^h) \subset B(p, \frac{1}{2}c_r\tau^h).$$

Now, let $q \notin P_{p^\ell}$ and $x^{\ell-1}$ be the ancestor of $q$ at level $\ell - 1$. Since $p^{\ell-1} \notin T$ and $T$ is semi-compressed, $d(p, x) > c_r\tau^{\ell-1}$. So,

$$d(p, q) \geq d(p, x) - d(x, q) \quad [\text{triangle inequality}]$$

$$> c_r\tau^{\ell-1} - \frac{c_c\tau}{\tau - 1} \tau^{\ell-1} \quad [\text{Lemma 3.1.1}]$$

$$\geq c_r\tau^{\ell-1} - \frac{c_r(\tau - 4)}{2(\tau - 1)} \tau^{\ell-1} \quad [\text{for } c_r \geq \frac{2c_c\tau}{\tau - 4}]$$

$$= \frac{\tau + 2}{2(\tau - 1)} c_r\tau^{\ell-1}$$

$$> \frac{1}{2}c_r\tau^{\ell-1}.$$ 

Notice that if $q$ does not have an ancestor in level $\ell - 1$, then the radius of the outer ball becomes even larger because $d(x, q) \leq \frac{c_c\tau}{\tau - 1} \tau^{\ell-2}$, see Fig. 4.8. 

In the following, we divide split touches into two categories of above and below. Also, we define two types of split events for the points in a permutation. Since these
Figure 4.8: Point $p_i$ with $C(p_i) = p_k^\ell$ in $P_{j-1}$, where $k < j < i$. The balls centered at $p_k$ from smaller to larger have radii $c_r \tau^h$, $\frac{c_r}{\tau-1} \tau^h$, $c_r \tau^{m-2}$, $c_r \tau^{m-1}$, $c_r \tau^m$, $\frac{1}{2} c_r \tau^{\ell-1}$, and $c_r \tau^{\ell-1}$, where $m := \lceil \log_{\tau}(d(p_i, p_k)/c_r) \rceil$. Left: a jump from $p_k^\ell$ to $p_k^h$ in a local net-tree on $P_{j-1}$. Center: the shaded annulus does not contain any points of $P_{j-1}$ and corresponds to the previous jump. Right: if $p_j$ causes a split below touch on $p_i$, then it is in the light shaded region. If $p_j$ causes a split above touch on $p_i$, then it is in the dark shaded region. The annuli defined by the two consecutive dashed balls are not necessarily empty in $P_{j-1}$.

Events only depend on the ordering of the points, and not the tree structure, we apply a backwards analysis to find the expected number of such events. Then, we show that these events can be used to bound the number of split touches.

Let $p_k^\ell := C(p_i)$ in $P_{j-1}$, where $k < j < i$, and $p_k^\ell$ be the top of a jump with the bottom node $p_k^h$, where $\ell \geq h + 2$ (by the definition of a jump). Also, let $m := \lceil \log_{\tau}(d(p_i, p_k)/c_r) \rceil$. By the definition of a center, $h + 1 \leq m \leq \ell$. Then, the insertion of $p_j$ at some level $f$, where $h \leq f \leq \ell - 1$, results that jump to be split at lower levels.

After splitting a jump, either $p_i$ stays in the same cell or it moves to the new cell of the new created node for $p_k$ in a lower level. When $p_i$ changes its center to the new node of $p_k$, we call that touch a split above. A split above touch implies that $p_j$ will no longer touch $p_i$ via a split touch. If $p_i$ remains in the cell of $p_k^\ell$, then we call that touch a split below, see Fig. 4.5b.
Now, we define two types of split events. A split below event \( \Psi_{i,j} \) is defined as follows. There is a bunch \( B \) near \( p_i \), for some constant values of \( \alpha \) and \( \beta \), and \( p_j \) is the unique farthest point in \( B \) to the first point in that bunch. Also we define a split above event \( \Phi_{ij} \) as follows. There is a bunch \( B \) near \( p_i \), for some constant values of \( \alpha \) and \( \beta \), and \( p_j \) is the unique closest point not in the bunch to the first point in \( B \). In the following lemmas, we find the expected number of these events.

**Lemma 4.3.12.** The expected number of split below events for a point \( p_i \) in a permutation \( \langle p_1, \ldots, p_n \rangle \) is \( O(\rho^{O(1)} \log n) \).

**Proof.** Let \( F(q, B) \) be a random event that \( q \in B \) proceeds all points of \( B \setminus \{q\} \). Also let \( B_1, \ldots, B_d \) be the bunches near \( p_i \) containing more than one point. By Lemma 4.3.10, \( d \) is a constant. So,

\[
Pr[\Psi_{i,j}] = \sum_{c=1}^{d} Pr[\Psi_{i,j} \mid p_j \in B_c] Pr[p_j \in B_c]
\]

\[
= \sum_{c=1}^{d} \left( \sum_{q \in B_c} Pr[\Psi_{i,j} \mid F(q, B_c)] Pr[F(q, B_c)] \right) Pr[p_j \in B_c]
\]

\[
\leq \sum_{c=1}^{d} \left( \sum_{q \in B_c} \frac{1}{|B_c| - 1} \cdot \frac{1}{|B_c|} \right) \frac{|B_c|}{j} \leq \frac{2d}{j}.
\]

Therefore,

\[
\sum_{j=1}^{i-1} Pr[\Psi_{i,j}] \leq 2d \sum_{j=1}^{n} \frac{1}{j} = O(\rho^{O(1)} \log n).
\]

**Lemma 4.3.13.** The expected number of split above events for a point \( p_i \) in a permutation \( \langle p_1, \ldots, p_n \rangle \) is \( O(\rho^{O(1)} \log n) \).

**Proof.** We define \( F(q, B) \) similar to the proof of Lemma 4.3.12. Let \( B_1, \ldots, B_d \) be the
bunches near \( p_i \). Then,

\[
Pr[\Phi_{i,j}] = \sum_{c=1}^{d} \sum_{q \in B_c} Pr[\Phi_{i,j} | F(q, B_c)] Pr[F(q, B_c)]
\]

\[
\leq \sum_{c=1}^{d} \sum_{q \in B_c} \left( \frac{1}{j - 1} \right) \left( \frac{1}{|B_c|} \right)
\]

\[
\leq \frac{d}{j - 1},
\]

which implies

\[
\sum_{j=2}^{i-1} Pr[\Phi_{i,j}] \leq d \sum_{j=1}^{n} 1/j = O(\rho^{O(1)} \log n).
\]

In the following, we show how split touches can be counted by the previous events.

**Theorem 4.3.14.** The expected number of split below touches in a random permutation is \( O(\rho^{O(1)} n \log n) \).

**Proof.** In order to relate split below touches to split below events, we should specify when such touches occur in a tree and then find the right bunches for the corresponding events. First, we show that if \( p_j \) touches \( p_i \) with a split below touch then either \( p_j \) belongs to a bunch near \( p_i \) or it is followed by a basic touch. We have \( h + 2 \leq m \leq \ell \), because for \( m = h+1 \) we always have split above touches (the cell of \( p_i \) will be changed).

Now, we have two cases: either \( m = \ell \) or \( h + 2 \leq m \leq \ell - 1 \).

(a) If \( m = \ell \), then the insertion of \( p_j \) at any level between \( h \) and \( \ell - 1 \) results a split below touch, i.e. \( h \leq f \leq \ell - 1 \). If \( f = \ell - 1 \), then \( p_j \) will also touch \( p_i \) with a basic touch, so we can charge the basic touch to pay the cost of the split below touch. If \( f = \ell - 2 \), then either \( p_j \) requires to be promoted to level \( \ell - 1 \) or it stays at the same level. If \( p_j \) is promoted to level \( \ell - 1 \), then it touches \( p_i \) with a basic touch, so we can charge the basic touch to pay the cost of the split below touch. If \( p_j \) stays at level \( \ell - 2 \), then no further promotion is needed for \( p_j \), which
implies that the highest level of $p_j$ after the insertion of all $n$ points will be $\ell - 2$. Therefore, each point may fall in this situation at most once, as such the total number of split below touches for all $n$ points falling in this category is $O(n)$.

Now, let $h \leq f \leq \ell - 3$. In this case, $h \leq \ell - 3 = m - 3$. From Lemma 4.3.11 and $c_r \tau^{m-1} < d(p_i, p_k) \leq c_r \tau^m$,

\[ P_{p_k^h} \subset B(p_k, \frac{1}{2} c_r \tau^h) \subset B(p_k, \frac{1}{2} c_r \tau^{m-3}) \subset B(p_k, \frac{1}{2\tau^2} d(p_i, p_k)) \]

and

\[ [B(p_k, \frac{1}{2\tau} d(p_i, p_k)) \setminus B(p_k, \frac{1}{2\tau^2} d(p_i, p_k))] \cap P_{j-1} = \emptyset. \]

Also,

\[ d(p_j, p_k) \leq c_r \tau^f \leq c_r \tau^{m-3} < \frac{1}{\tau^2} d(p_i, p_k). \]

Since for $\tau > 2$, we have $\frac{1}{\tau^2} < \frac{1}{\tau^4} < \frac{1}{\tau^2}$, if we set $\alpha = \frac{1}{\tau}$ and $\beta = \frac{1}{\tau^2}$ in Definition 4.3.8, then $p_j$ will be in a bunch near $p_i$.

(b) If $h + 2 \leq m \leq \ell - 1$, then the insertion of $p_j$ at any level between $h$ and $m - 2$ results a split below touch, i.e. $h \leq f \leq m - 2$, see Fig. 4.8. Using Lemma 4.3.11 and $c_r \tau^{m-1} < d(p_i, p_k) \leq c_r \tau^m$,

\[ P_{p_k^h} \subset B(p_k, \frac{1}{2} c_r \tau^h) \subset B(p_k, \frac{1}{2} c_r \tau^{m-2}) \subset B(p_k, \frac{1}{2\tau} d(p_i, p_k)) \]

and

\[ [B(p_k, \frac{1}{2} d(p_i, p_k)) \setminus B(p_k, \frac{1}{2\tau} d(p_i, p_k))] \cap P_{j-1} = \emptyset. \]

Also,

\[ d(p_j, p_k) \leq c_r \tau^f \leq c_r \tau^{m-2} < \frac{1}{\tau} d(p_i, p_k). \]
For $\tau > 2$, we have $\frac{1}{2\tau} < \frac{1}{\tau} < \frac{1}{2}$. Therefore, if we set $\alpha = \frac{1}{\tau}$ and $\beta = \frac{1}{2}$ in Definition 4.3.8, then $p_j$ belongs to a bunch near $p_i$.

So far, we proved that if $p_j$ causes a split below touch on $p_i$ and is not followed by a basic touch, then $p_j$ is in a bunch near $p_i$. However, $p_j$ is not necessarily the farthest point in the bunch. If $d(p_j, p_k) > \frac{c_c}{\tau - 1}\tau^{h+1}$, then by Lemma 3.1.1, $p_j$ is the unique farthest point to $p_k$. If $d(p_j, p_k) \leq c_c\tau^h$, then $p_j$ is in the subtree rooted at $p_k$ and should not be promoted to a level greater than $h - 1$, so $p_j$ will never results a split touch on $p_i$. The remaining case is when $c_c\tau^h < d(p_j, p_k) \leq c_c/(\tau - 1)\tau^{h+1}$, see Fig. 4.8.

For all points in $P_j$ of a distance in $(c_c\tau^h, \frac{c_c}{\tau - 1}\tau^{h+1}]$ from $p_k$, only one can touch $p_i$ via a split below touch, because the first point creates a node $p_k^{h+1}$ and touches $p_i$ and the remaining will be added as children of $p_k^{h+1}$, so they will not touch $p_i$ with a split below touch. In this case, we charge the farthest point to pay the cost of this split below touch. This argument is valid only if $p_k^{h+1}$ is not removed later, and this removal happens when the only child of $p_k^{h+1}$ finds a closer parent. By the triangle inequality, we can easily show that $p_k^{h+1}$ and the new parent are relatives, so $p_k^{h+1}$ should remain in the tree. Thus, either the farthest point touches $p_i$ via a split below touch or it pays for another point that touches $p_i$.

Also note that the order of points in a permutation determines the first point of a bunch, and it is important because one of its associated nodes is the center of $p_i$ in $P_{j-1}$. Furthermore, using Lemma 4.3.3, the maximum number of split below touches from $p_j$ on $p_i$ is constant. In conclusion, the expected number of split below touches on $p_i$ is bounded by the summation of the expected number of split below events (Lemma 4.3.12) and basic touches (Theorem 4.3.7).

**Theorem 4.3.15.** The expected number of split above touches in a random permutation is $O(\rho^{O(1)}n \log n)$. 

77
Proof. Our aim is to make a connection between split above touches and events. For a split above touch we have $h + 1 \leq m \leq \ell - 1$ and $m - 1 \leq f \leq \ell - 1$, see Fig. 4.8. If $h > -\infty$, then the subtree rooted at $p_k^h$ has more than one point. Therefore, before $p_j$ touches $p_i$ via a split above touch, $p_i$ should have been touched with a split below touch by a node in level $h - 1$. We charge that split below to pay the cost of split above touches on $p_i$ for levels $h$, $h + 1$, and $h + 2$. In other words, if $h \leq f \leq h + 2$, then the cost of split above touches has been already paid by an earlier split below touch. As such, we only need to handle the remaining split above touches when $f \geq h + 3$ and $h + 4 \leq m \leq \ell - 1$.

Using Lemma 4.3.11 and $c_r \tau^{m-1} < d(p_i, p_k) \leq c_r \tau^m$,

$$P_{p_k}^h \subset B(p_k, \frac{1}{2} c_r \tau^h) \subset B(p_k, \frac{1}{2} c_r \tau^{m-1}) \subset B(p_k, \frac{1}{2} \tau^3 d(p_i, p_k))$$

and

$$[B(p_k, \frac{1}{2} d(p_i, p_k)) \setminus B(p_k, \frac{1}{2} \tau^3 d(p_i, p_k))] \cap P_{j-1} = \emptyset.$$ 

If $p_j$ is promoted to level $f$, then $p_j^{f-1}$ violates the covering property and as such,

$$d(p_j, p_k) > c_r \tau^f \geq c_r \tau^{m-1} = \frac{\tau - 4}{2 \tau} c_r \tau^{m-1} \geq \frac{1}{10 \tau} c_r \tau^m \geq \frac{1}{10 \tau} d(p_i, p_k).$$

Otherwise, if $p_j$ is directly inserted into level $f$, $p_j$ cannot be a relative of $p_k$ at level $f - 1$, as such

$$d(p_j, p_k) > c_r \tau^{f-1} \geq c_r \tau^{m-2} \geq \frac{1}{\tau^2} d(p_i, p_k).$$

Since $\tau \geq 5$,

$$d(p_j, p_k) > \min\{\frac{1}{10 \tau}, \frac{1}{\tau^2}\} d(p_i, p_k) \geq \frac{1}{2 \tau^2} d(p_i, p_k).$$

Also, we have $\frac{1}{2 \tau^3} < \frac{1}{2 \tau^2} < \frac{1}{2}$, so if we set $\alpha = \frac{1}{2 \tau^3}$ and $\beta = \frac{1}{2 \tau^2}$ in Definition 4.3.8, then
$p_j$ does not belong to any bunch near $p_i$.

From Lemma 4.3.11, the distance of every point of $P_{j-1}$ not in $P_{P_k^h}$ to $p_k$ is greater than $\frac{1}{2}c_r\tau^{\ell-1}$. We know that if $p_j$ touches $p_i$ via a split above touch, then $d(p_j, p_k) \leq c_r\tau^{\ell-1}$. So, if $d(p_j, p_k) \leq \frac{1}{2}c_r\tau^{\ell-1}$, then $p_j$ is the closest point to $p_k$ not in that bunch, as desired. Otherwise, if $\frac{1}{2}c_r\tau^{\ell-1} < d(p_j, p_k) \leq c_r\tau^{\ell-1}$, we use a charging argument similar to the proof of Theorem 4.3.14. In other words, if there are many points of a distance in $(\frac{1}{2}c_r\tau^{\ell-1}, c_r\tau^{\ell-1}]$ from $p_k$, then only one of them results a split above touch, and it is the first one that splits the jump at level $\ell - 1$. Therefore, we can charge the closest point to $p_k$ not in the bunch to pay the cost of that split above touch. Notice that because $p_k^{\ell-1}$ is relative to the point that creates it, $p_k^{\ell-1}$ will not be removed later and will be in the output.

For $h = -\infty$, i.e. the subtree rooted at $P_k^h$ only contains $p_k^h$, we can use a similar argument (without charging split below touches). In a nutshell, the expected number of split above touches on $p_i$ is bounded by the expected number of split above events (Lemma 4.3.13) and split below touches (Theorem 4.3.14) on $p_i$.

**Theorem 4.3.16.** The expected number of merge touches in a random permutation is $O(\rho^{O(1)}n \log n)$.

**Proof.** Recall that the insertion of $p_j$ results a merge touch on $p_i$ if $p_k^\ell := C(p_i)$ is deleted from the tree and $p_i$ moves to the cell of $\text{par}(p_k^\ell)$. Let $p_k^\ell$ be added to the tree after the insertion of a point $p_a$, where $a < j$. Since $p_i$ is in the cell of $p_k^\ell$, $p_i$ belongs to the cell of $\text{par}(p_k^\ell)$ before $p_a$ is inserted. Therefore, the insertion of $p_a$ results a split above touch on $p_i$. In other words, before $p_j$ touches $p_i$ via a merge touch, there exists another point $p_a$, where $a < j$, such that it touches $p_i$ via a split above touch. Therefore, it suffices to pay the cost of a merge touch when an earlier split above touch occurs. Thus, the total number of merge touches is bounded by the total number of split above touches (Theorem 4.3.15).
4.4 Experimental Results

The analysis of our net-tree algorithm involves several tradeoffs, constants, and packing arguments that produce constant factors that are probably not tight in practice. In this section, we use several toy examples to analyze the performance of our algorithms. We selected data sets with certain geometric properties to try to identify the impact of factors such as dimension and the spread on the running time.

We implemented our net-tree construction algorithms in about 300 lines of Python [47], which is a testament to their simplicity. Instead of looking at the construction times, we use the number of required distance computations as a more meaningful measure for comparing algorithms. It resembles the comparison model for sorting algorithms. Therefore, the practical performance of our algorithms will be independent of the machine used for experiments. Moreover, for high dimensional spaces or spaces with complex distance functions, each distance computation is costly, and the actual construction times will be dominated by the number of distance computations. We use the Euclidean distance as a metric and examine our algorithms on the following data sets.

- **DS1.** Points uniformly sampled from a hypercube in $\mathbb{R}^3$.

- **DS2.** Points uniformly sampled from a line between 0 and 1.

- **DS3.** Points sampled from a standard normal distribution in $\mathbb{R}^3$.

- **DS4.** Points $\{p_1, p_2, \cdots, p_n\}$ on a line such that for each point $p_i$, its coordinate is $2^i$.

- **DS5.** Three points with fixed coordinates $(0, 0, 0)$, $(1, 0, 0)$, and $(2^n, 0, 0)$. The remaining points sampled uniformly from a cube in $\mathbb{R}^3$. 
Note that while the last two data sets provide exponential spreads, DS5 uses a uniform distribution. We choose the number of points in our experiments from the following set

\{100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000\}.

Since the metric for DS3 is not costly, we run more experiments for data sets of sizes ranging from 11000 to 30000 with the step size of 1000. Also, we set \(c_c = c_p = 1\) and change the scale factor \(\tau\) from 5 to 10 to observe which value works better in practice.

We will define several metrics in Section 4.4.1 that will be used later to compare the performance of our algorithms. Then, in Section 4.4.2, we will compare the practical performance of different point location strategies introduced in this chapter. We will also discuss some lessons learned from the experiments regarding which algorithm works best for which types of data.

### 4.4.1 Model Comparison Metrics

In this section, we define a set of metrics to compare the practical performance of our algorithms.

**Distance Increase Rate**

We use this metric to examine the time complexity of our construction algorithm with eager point location (described in Section 4.3.1) in practice. We define the distance increase rate from a smaller point set \(A\) to a larger point set \(B\) to be the ratio of the number of distance computations required in \(B\) to the number of distance computations used in \(A\). Note that in our algorithms we may recalculate some distances many times. In this metric, we count all distance computations, not just distinct computations.
We proved that the expected time complexity of our randomized incremental algorithm is $O(n \log n)$. If we test our algorithm for a data set $A$ with $n$ points and another data set $B$ with $kn$ points with the same distribution, then we expect the following distance increase rate assuming $A$ and $B$ having the same doubling dimension:

$$\frac{O(kn \log kn)}{O(n \log n)} \approx k\left(\frac{\log k + \log n}{\log n}\right) = k(1 + \log_n k).$$

(4.3)

For example, the expected distance increase rate from 1000 to 2000 points is $2(1 + \log_{1000} 2) \approx 2.2$. Figs. 4.9, 4.10, 4.11, 4.12, and 4.13 compare the observed distance increase rate of the aforementioned data sets with their theoretical expected values obtained from (4.3). As can be seen in the figures, for different values of $\tau$, the increase rate is close to the theoretical bound, which implies that the doubling dimension is stable and they reached the asymptotic regime.

**Speedup**

We are also interested to compare the practical performance of different point location schemes introduced in this chapter. We compare the eager point location against the two lazy point location algorithms presented in Section 4.2.4. This comparison will be based on the speedup gained from replacing a lazy point location with the eager point location algorithm. We define the speedup to be the ratio of the number of distance computations required to build a net-tree with a single path or multipath point location to the number of distance computations needed to build a net-tree with the eager point location. Similar to the distance increase rate, the speedup also counts the repeated distance calculations as multiple calculations.

Figs. 4.14, 4.15, 4.16, 4.17, and 4.18 illustrate the speedup for different data sets. For all the values of $\tau$, single path point location is faster than multipath point location.
Figure 4.9: Comparing the distance increase rate for DS1 with different values of the scale factor $\tau$. 
Figure 4.10: Comparing the distance increase rate for DS2 with different values of the scale factor $\tau$. 
Figure 4.11: Comparing the distance increase rate for DS3 with different values of the scale factor $\tau$. 
Figure 4.12: Comparing the distance increase rate for DS4 with different values of the scale factor $\tau$. 

---

86
Figure 4.13: Comparing the distance increase rate for DS5 with different values of the scale factor $\tau$. 
Also, the gained speedup for all the cases is always greater than 1, which indicates that the eager point location is faster than its lazy counterparts.

**Basic Touch Percentage**

In the next set of experiments, we examine the percentage of basic touches in the eager point location scheme. This percentage simply indicates what fraction of touches are basic. Fig. 4.19 shows this percentage for different values of $\tau$ for all of the data sets.

**Unique Distance Computations**

In this metric, we determine what fraction of total distances are used in our randomized incremental construction algorithm. Note that for a data set of size $n$, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ distances. In our algorithm, we may recalculate the distance between two points many times. We can use a hash table to cache the distance of every two points needed in our algorithm to eliminate the cost of recalculation, but it comes at the cost of increased space. We define the unique distance percentage to be percentage of distances (ignoring recalculations) required in our construction method. This metric is simply equal to the percentage of the $\binom{n}{2}$ distances stored in the hash table. Fig. 4.20 illustrates this metric for different data sets.

### 4.4.2 Discussion

In our experiments, we observed that by increasing the value of $\tau$ from 5 to 6, we get a considerable improvement in the number of distance computations and it is because of the value of the relative constant $c_r$. In our experiments, we set $c_r = \frac{2c_c\tau}{(\tau - 4)}$ and $c_c = c_p = 1$. As a result, for $\tau = 5$ and $\tau = 6$, we have $c_r = 10$ and $c_r = 6$, respectively. This means that in lower levels of net-trees, which contain more nodes, the number of relative links for $\tau = 5$ is much larger than for $\tau = 6$. Therefore, the
Figure 4.14: Comparing the speedup for DS1 with different values of the scale factor $\tau$
Figure 4.15: Comparing the speedup for DS2 with different values of the scale factor $\tau$. 
Figure 4.16: Comparing the speedup for DS3 with different values of the scale factor $\tau$. 
Figure 4.17: Comparing the speedup for DS4 with different values of the scale factor $\tau$. 
Figure 4.18: Comparing the speedup for DS5 with different values of the scale factor \( \tau \).
Figure 4.19: The percentage of basic touches for different values of $\tau$ on different data sets.
Figure 4.20: The percentage of unique distance computations in the randomized incremental construction algorithm for different values of $\tau$ on different data sets.
number of distance computations required in point location decreases by increasing the scale factor from 5 to 6.

Based on Fig. 4.17, the eager point location scheme for data set DS4 results in a huge speedup over the lazy schemes. We justify this observation in the following. Since the searches for the center in the lazy point location schemes start from the root, their running times depend mostly on the number of levels in the net-tree. From Table 4.1, we can observe that the number of levels for DS4 is a linear function of the number of points. As such, the cost of point location in the lazy approaches is linear for each point. On the other hand, the eager point location does not have such dependence on the number of levels. We can use a similar argument to justify why the speedup for DS2 is more than other data sets.

Another interesting observation is related to Fig. 4.20. For 10000 points, the randomized incremental construction on DS2 and DS4 only uses less than 2 percent and 0.2 percent of total distances, respectively. In other words, our construction algorithm for DS2 and DS4 runs at least 10 times and 100 times faster than for other data sets, respectively. According to Fig. 4.19, almost all touches for DS1, DS3, and DS5 are basic. However, this metric for DS2 and DS4 is about 92 percent and 20 percent, respectively. This observation is consistent with our understanding of the eager point location behavior. Recall in our eager point location algorithm, if the insertion of a new node results in a split or merge event, we only need to touch the points in the Voronoi cell of a single node. Otherwise, we should touch the points in nearby Voronoi cells of nodes from one level up to one level down. Therefore, the efficiency of our construction algorithm has a direct relationship with the fraction of split and merge touches. Moreover, we get more split or merge touches if the generated net-tree has more levels.

In summary, we observed in our experiments that the eager point location approach
Table 4.1: The number of levels and jumps (excluding $-\infty$ and $+\infty$) in net-trees generated by the randomized incremental algorithm with $\tau = 6$.

works better than the others. Also, our construction algorithm runs especially fast on data sets for which the corresponding net-trees are deep.
<table>
<thead>
<tr>
<th>Points</th>
<th></th>
<th>Nodes</th>
<th></th>
<th>Relatives</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DS1</td>
<td>DS2</td>
<td>DS3</td>
<td>DS4</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>117</td>
<td>158</td>
<td>126</td>
<td>175</td>
</tr>
<tr>
<td>500</td>
<td></td>
<td>633</td>
<td>829</td>
<td>585</td>
<td>885</td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>1326</td>
<td>1614</td>
<td>1231</td>
<td>1767</td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td>2767</td>
<td>3235</td>
<td>2549</td>
<td>3541</td>
</tr>
<tr>
<td>3000</td>
<td></td>
<td>4099</td>
<td>4867</td>
<td>3915</td>
<td>5316</td>
</tr>
<tr>
<td>4000</td>
<td></td>
<td>5323</td>
<td>6477</td>
<td>5223</td>
<td>7091</td>
</tr>
<tr>
<td>5000</td>
<td></td>
<td>6477</td>
<td>8098</td>
<td>6572</td>
<td>8862</td>
</tr>
<tr>
<td>6000</td>
<td></td>
<td>7595</td>
<td>9785</td>
<td>7882</td>
<td>10637</td>
</tr>
<tr>
<td>7000</td>
<td></td>
<td>8725</td>
<td>11384</td>
<td>9121</td>
<td>12407</td>
</tr>
<tr>
<td>8000</td>
<td></td>
<td>9849</td>
<td>13071</td>
<td>10413</td>
<td>14182</td>
</tr>
<tr>
<td>9000</td>
<td></td>
<td>10918</td>
<td>14675</td>
<td>11654</td>
<td>15957</td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td>12021</td>
<td>16282</td>
<td>12878</td>
<td>17732</td>
</tr>
</tbody>
</table>

Table 4.2: The number of nodes and relatives (excluding self relatives) in net-trees generated by the randomized incremental algorithm with $\tau = 6$. 

98
Chapter 5

Net-Trees and Permutations

Greedy permutations are one of the most famous point orderings that have been used for clustering and sampling applications. Greedy permutations are very closely related to net-trees. Har-Peled and Mendel [41] showed how a net-tree can be constructed from a greedy ordering in linear time. In this chapter, we introduce two wider class of permutations called approximate greedy permutations and locally greedy permutations. Then, we present two linear time algorithms to build net-trees from these permutations and vice versa.

In Section 5.2.1, we prove that the point location step of the net-tree construction algorithm only requires $O(n)$ time if the points are given in an approximate greedy ordering. Therefore, if the input points are ordered in an approximate greedy form, then we can construct a net-tree from that permutation in linear time. In Section 5.2.2, we show how a locally greedy permutation can be constructed from a given net-tree in linear time.
5.1 Greedy Permutations and Their Relatives

A permutation \( \pi \) for a point set \( P \) with \( n \) points is an ordering \( \langle p_1, \ldots, p_n \rangle \) of points of \( P \). Let \( P_i := \{ p_1, \ldots, p_i \} \) be the \( i \)-th prefix of \( \pi \) for \( 1 \leq i \leq n \). We define the predecessor of a point \( p_i \) in \( \pi \) as the closest point to \( p_i \) in \( P_{i-1} \) and we denote it by \( \text{pred}(p_i) \). For point \( p_1 \), let \( \text{pred}(p_1) \) be empty. We define the insertion radius of \( p_i \) to be \( \lambda_i := d(p_i, \text{pred}(p_i)) \). We set \( \lambda_1 = +\infty \).

Definition 5.1.1. A greedy permutation is a permutation \( \pi \) such that for each \( i > 1 \) and \( p_i \in \pi \),
\[
d(p_i, P_{i-1}) = d_H(P_{i-1}, P).
\]

The greedy permutation is also called the farthest point sampling. This definition suggests a naive \( O(n^2) \) time algorithm for the greedy permutation problem [35]. Har-Peled and Mendel [41] proposed an \( O(n \log n) \) algorithm to find (approximate) greedy permutations in doubling spaces.

Definition 5.1.2. For \( \delta \geq 1 \), a \( \delta \)-approximate greedy permutation is a permutation \( \pi \) such that for each \( i > 1 \) and \( p_i \in \pi \), we have
\[
\frac{d_H(P_{i-1}, P)}{\delta} \leq d(p_i, P_{i-1}) \leq d_H(P_{i-1}, P).
\]

Approximate greedy permutations are more general than greedy permutations, since a greedy permutation is 1-approximate greedy. Our notion of approximate greedy permutations is also more general than of Eppstein et al. [31]. They proposed randomized algorithms to find \((1+\varepsilon)\)-approximate greedy permutations for both graph metrics and high dimensional Euclidean spaces. The following lemmas relate these permutations to metric nets.

Lemma 5.1.3. In a greedy permutation, each \( P_i \) is a \( (\lambda_i, \lambda_{i+1}) \)-net.
Proof. By the definition of the greedy permutation, no point of \( P \setminus P_i \) can be farther than \( \lambda_{i+1} \). The insertion radii of \( p_1, \ldots, p_n \) are non-increasing. So, no two points can be closer than the insertion radius of the latter, and thus, no pair in \( P_i \) can be closer than \( \lambda_i \).

Lemma 5.1.4. Given \( \delta \geq 1 \), each \( P_i \) in a \( \delta \)-approximate greedy permutation is a \( \left( \frac{\lambda_i}{\delta}, \min\{\lambda_1, \ldots, \lambda_{i+1}\} \delta \right) \)-net.

Proof. By the definition of a \( \delta \)-approximate greedy permutation,

\[
\lambda_i = d(p_i, P_{i-1}) \geq \frac{1}{\delta} d_H(P_{i-1}, P).
\]

So, \( d_H(P_i, P) \leq \delta \lambda_{i+1} \). Also,

\[
d_H(P_i, P) \leq d_H(P_{i-1}, P) \leq \delta \lambda_i.
\]

If we continue, we get \( \beta = \min\{\lambda_1, \ldots, \lambda_{i+1}\} \delta \).

On the other hand, assume that for two points \( p_i, p_j \in \pi, 1 < j < i \). So,

\[
\lambda_i = d(p_i, P_{i-1}) \leq d(p_i, P_{j-1}) \leq \max_{p \in P} d(p, P_{j-1}) = d_H(P, P_{j-1}) \leq \delta d(p_j, P_{j-1}) \leq \delta \lambda_j.
\]

Therefore, \( \alpha = \lambda_i / \delta \).

Now, we define a much wider class of permutations, called locally greedy permutations. Before, we define the aspect ratio of a point. Given \( Q \subseteq P \), the aspect ratio of
a point $p \in Q$ with respect to $Q$ is

$$
\text{aspect}_Q(p) := \max_{q \in \text{Vor}_Q(p)} \frac{d(p, q)}{d(p, Q \setminus \{p\})}.
$$

In this definition, $\text{Vor}_Q(p)$ is the Voronoi cell of $p$, i.e.

$$
\text{Vor}_Q(p) := \{x \in P \mid d(p, x) = d(x, Q)\},
$$

see Fig. 5.1. Note that the aspect ratio can be less than one. For example, when

Figure 5.1: The points are ordered from $p_1$ to $p_{12}$. The black dots show the prefix $P_7$ and white dots are $P \setminus P_7$. The Voronoi diagram for $P_7$ is illustrated by the solid edges. For point $p_2$, $p_6$ is its nearest neighbor in $P_7$. Also, $p_{10}$ is the farthest point in the Voronoi cell of $p_2$. Therefore, the aspect ratio of $p_2$ with respect to $P_7$ is $\frac{d(p_2, p_{10})}{d(p_2, p_6)}$.

$Q = P$, the aspect ratio of every point is zero. The aspect ratio of $Q \subseteq P$ is the maximum aspect ratio of all points in $Q$. Similarly, the aspect ratio of a permutation $\pi = \langle p_1, \ldots, p_n \rangle$ of $P$ is the maximum aspect ratio among all prefixes of $\pi$. More
formally,
\[
\text{aspect}(\pi) := \max_{i \in \{1, \ldots, n\}} \max_{p \in P_i} \{\text{aspect}_{P_i}(p)\}.
\]

In the following, we show that the aspect ratio of greedy and approximate greedy permutations are bounded.

**Lemma 5.1.5.** The aspect ratio of a greedy permutation is at most one.

**Proof.** From Lemma 5.1.3, the aspect ratio is
\[
\frac{\lambda_{i+1}}{\lambda_i} \leq \frac{\lambda_i}{\lambda_i} = 1.
\]

**Lemma 5.1.6.** A $\delta$-approximate greedy permutation has the aspect ratio of at most $\delta$.

**Proof.** From Definition 5.1.2, the aspect ratio is
\[
\frac{d_H(P_i, P)}{d_H(P_i-1, P) / \delta} \leq \delta \frac{d_H(P_i, P)}{d_H(P_i, P)} = \delta.
\]

Note that the notion of bounded aspect ratio permutations is inherently local, so the greedy permutation is overkill because it involves a global optimization. Instead, we consider a more local definition. In the following, we define locally greedy permutations.

**Definition 5.1.7.** Given a constant $\delta \geq 0$, a permutation $\pi$ is a $\delta$-locally greedy if and only if $\text{aspect}(\pi) \leq \delta$.

From Definition 5.1.7 and Lemma 5.1.6, every $\delta$-approximate greedy permutation is a $\delta$-locally greedy. However, the converse is not necessarily true. Fig. 5.2 illustrates the relationship between different permutations introduced in this section.
5.2 Transition between Greedy Permutations and Net-Trees

In this section, we propose linear time algorithms to build a net-tree from a locally greedy permutation and vice versa.

5.2.1 Net-Trees from Locally Greedy Permutations

As shown by Har-Peled and Mendel [41], following the example of Clarkson [21], a net-tree can be constructed in linear time from a greedy permutation assuming that one also has the nearest (or approximate nearest) neighbor of each point among its predecessors. This pairing of points with their predecessors is a natural byproduct of many (if not most) algorithms for computing greedy permutations. One benefit of this perspective is that it clarifies which aspects of the construction can be done in linear time and which take superlinear time.
Theorem 5.2.1. Given a $\delta$-locally greedy permutation of a set $P$ of $n$ points and a predecessor pairing. A net-tree $T \in \text{LNT}(\tau, c_p, c_c)$ for $P$ with $c_r = \frac{2c_r \tau}{\tau - 2}$ can be constructed in $O(\rho^{O(1)} n \log_\tau \delta)$ time, where $\delta > 0$ is a constant.

Proof. To prove this theorem, we propose a constant time point location algorithm for each inserting point. For a new point $q$, starting from its nearest neighbor $p$ among all inserted points, consider the node $p^\ell := \text{par}(p^\rightarrow \infty)$. If $d(p, q) \leq c_r \tau^\ell$, then $p^\ell$ is the center of $q$. Otherwise, search up the tree to find the center. In this search, if the closest node to $q$ among rel(par($p^\ell$)), say $x^h$, can serve as the center, then we stop, otherwise, the search continues with $x^h$. It is easy to see that this search always terminates because the root can serve as the ultimate center.

Let $p_i$ be a point in a given $\delta$-locally greedy permutation with $p_j := \text{pred}(p_i)$, where $j < i$. Also the aspect ratio of $p_j$ in $P_{i-1}$ is at most $\delta$, so

$$\delta \geq \frac{\max_{q \in \text{Var}_{P_{i-1}}(p_j)} d(p_j, q)}{d(p_j, P_{i-1} \setminus \{p_j\})} \geq \frac{d(p_j, p_i)}{d(p_j, P_{i-1} \setminus \{p_j\})}.$$ 

Therefore,

$$d(p_i, p_j) = d(p_i, P_{i-1}) \leq \delta d(p_j, P_{i-1} \setminus \{p_j\}).$$

Using Lemma 4.1.1 and the above inequality,

$$d(p_i, C(p_i)) \leq 2(\tau - 1)d(p_i, P \setminus \{p_i\}) \leq 2(\tau - 1)d(p_i, P_{i-1}) \leq 2\delta(\tau - 1)d(p_j, P_{i-1} \setminus \{p_j\}).$$

Let $p_j^\ell = \text{par}(p_j^\rightarrow \infty)$. Then, either $p_j^\ell$ has a relative besides itself or a sibling. Let $p_k^\ell$
represent this node. Then,

$$d(p_j, p_k) \geq d(p_j, P_{i-1} \setminus \{p_j\}).$$

Combining the last two inequalities results in

$$d(p_i, C(p_i)) < 2\delta(\tau - 1)max\{c_r \tau^\ell, c_c \tau^{\ell+1}\}.$$ 

Let $m = \lceil \log_\tau(d(p_i, C(p_i))/c_r) \rceil$. Then,

$$c_r \tau^{m-1} < d(p_i, C(p_i)) \leq c_r \tau^m.$$ 

The last two inequalities enable us to obtain an upper bound on the number of levels that the point location algorithm requires to climb. In other words, $m - \ell = O(\log_\tau \delta)$, which implies that the cost of point location for each point will be constant.

### 5.2.2 Locally Greedy Permutations from Net-Trees

In this section, we present a linear time algorithm to extract a locally greedy permutation from a net-tree. We can use the refining operation introduced in Section 3.3.2 to decrease the scale factor of the net-tree, and as a result, reduce the aspect ratio of the generated locally greedy permutation.

**Theorem 5.2.2.** Let $T$ be a semi-compressed net-tree in $\text{LNT}(\tau, c_p, c_c)$. A $\left(\frac{c_r \tau^2}{c_p(\tau-1)}\right)$-locally greedy permutation can be extracted from $T$ in $O(\rho^{O(1)} n)$ time.

**Proof.** We create the permutation by traversing the nodes of $T$ level by level. In the
following, we prove that the aspect ratio of the resulted permutation is \( \frac{c_c \tau^2}{c_p(\tau-1)} \) by induction. Note that we can slightly change our data structure so that the root exists at one level above the highest level of the \( T \) except \( +\infty \).

At the beginning, \( p_1 \) is the root and its aspect ratio is 0. For \( i > 1 \), let \( P_i \) be \( (\frac{c_c \tau^2}{c_p(\tau-1)}) \)-locally greedy. We prove that \( P_{i+1} \) is also locally greedy. Let \( p_i \) be added to the permutation by node \( p^\ell_i \). Then, point \( p_{i+1} \) has a node either at level \( \ell \) or \( \ell - 1 \) such that the algorithm visits it after \( p^\ell_i \). If \( p_{i+1} \) is added by a node at level \( \ell \), then by Lemma 3.1.1, \( d_H(P_{i+1}, P) \leq \frac{c_c \tau}{\tau-1} \tau^{\ell+1} \) because \( p_{i+2} \) may also have an associated node at level \( \ell \). The packing property results \( d(p_{i+1}, P) > c_p \tau^\ell \). So,

\[
\frac{d_H(P_{i+1}, P)}{d(p_{i+1}, P)} < \frac{\frac{c_c \tau}{\tau-1} \tau^{\ell+1}}{c_p \tau^\ell} = \frac{c_c \tau^2}{c_p(\tau-1)}.
\]

If \( p_{i+1} \) is added by a node at level \( \ell - 1 \), then the algorithm has visited all the nodes at level \( \ell \), and by Lemma 3.1.1, \( d_H(P_{i+1}, P) \leq \frac{c_c \tau}{\tau-1} \tau^\ell \). Also, by the packing property, \( d(p_{i+1}, P) > c_p \tau^{\ell-1} \). So,

\[
\frac{d_H(P_{i+1}, P)}{d(p_{i+1}, P)} < \frac{\frac{c_c \tau}{\tau-1} \tau^\ell}{c_p \tau^{\ell-1}} = \frac{c_c \tau^2}{c_p(\tau-1)}.
\]

Therefore, the generated permutation is both locally and approximate greedy. Since there are \( O(\rho^{O(1)}n) \) nodes in \( T \), the time complexity will be linear in terms of the number of nodes.
Chapter 6

Hierarchical Metric Trees for
Topological Data Analysis

In this chapter, we show how hierarchical metric trees can be applied to the topological data analysis. The goal of topological data analysis is finding the underlying shape of a data set. Given a finite data set in a Euclidean space, it is natural to consider the balls around the data points as a way to fill in the space around the data and give an estimate of the missing data. The union of balls is often called the offsets of the point set.

Persistent homology was originally invented as a way to study the changes in topology of the offsets of a point set as the radius increases from 0 to ∞. The input to persistent homology is usually a filtered simplicial complex, that is, an ordered collection of simplices such that each simplex appears only after its boundary simplices of one dimension lower. The Nerve Theorem and its persistent variant allow one to compute the persistent homology of the offsets by instead looking at a discrete object,

Some results in this chapter are published in [15, 16].
a filtered simplicial complex called the nerve (see Fig. 6.1). The simplest version of this complex is called the Čech complex and it may be viewed as the set of all subsets of the input, ordered by the radius of their smallest enclosing ball. Naturally, the Čech complex gets very big very fast, even when restricting to subsets of constant size. A common alternative is the Rips complex but it suffers similar difficulties. Over the last few years, there have been several approaches to building sparser complexes that still give good approximations to the persistent homology [63, 51, 28, 12, 11].

In this chapter, we present a simpler explanation for the construction and proof of correctness of sparse filtrations. Our new geometric construction shows that the sparse complex is just a nerve in one dimension higher. The approach easily generalizes to Rips, Čech and related complexes (the offsets for any convex metric). This is another advantage of the geometric view as the main result follows from convexity rather than explicit construction of simplicial map homotopy equivalences.

Furthermore, using hierarchical metric trees, we propose efficient algorithms for computing the edges of a sparse filtration in linear time, their birth times, and subsequently the $k$-simplices from a greedy permutation.

We also present a simple geometric proof that the explicit removal of vertices from
the sparse filtration can be done with edge contractions. This can be done without resorting to the full-fledged zigzag persistence algorithm [14, 13, 56, 57] or even the full simplicial map persistence algorithm [28, 10].

In Section 6.1, we present a perturbation scheme that reduces the complexity of offsets as the radius grows. We sparsify the perturbed offsets and show that the persistence barcode of the sparse nerve filtration is an $\varepsilon$-approximation to the persistence barcode of the offsets in Section 6.2. We present an algorithm in Section 6.3 to find all simplices of the sparse filtration in linear time from a given greedy permutation. Finally, in Section 6.4 we show that although vertices are not removed in the standard sparse filtration, they can be removed using edge contractions.

### 6.1 Perturbed Distances

A convenient first step in making a sparse version of the Čech filtration is to “perturb” the distance. Given a greedy permutation, we perturb the distance function so that as the radius increases, only a sparse subset of points continues to contribute to the offsets. This can most easily be viewed as changing the radius of the balls slightly so that some balls will be completely covered by their neighbors and thus will not contribute to the union. Fix a constant $\varepsilon < 1$ that will control the sparsity. As we will show in Lemma 6.1.1, at scale $\alpha$, there is an $\varepsilon\alpha$-net of $P$ whose perturbed offsets cover the perturbed offsets of $P$. Assuming the points $P = \{p_1, \ldots, p_n\}$ are ordered by a greedy permutation with insertion radii $\lambda_1, \ldots, \lambda_n$, we define the radius of $p_i$ at scale $\alpha$ as

$$r_i(\alpha) := \begin{cases} 
\alpha & \text{if } \alpha \leq \frac{\lambda_i(1+\varepsilon)}{\varepsilon} \\
\frac{\lambda_i(1+\varepsilon)}{\varepsilon} & \text{otherwise.}
\end{cases}$$
The perturbed $\alpha$-offsets are defined as

$$\tilde{P}^\alpha := \bigcup_{i \in [n]} B(p_i, r_i(\alpha)).$$

To realize the sparsification as described, we want to remove balls associated with some of the points as the scale increases. This is realized by defining the $\alpha$-ball for a point $p_i \in P$ to be

$$b_i(\alpha) := \begin{cases} B(p_i, r_i(\alpha)) & \text{if } \alpha \leq \frac{\lambda_i(1+\varepsilon)^2}{\varepsilon} \\ \emptyset & \text{otherwise.} \end{cases}$$

The usefulness of this perturbation is captured by the following covering lemma, which is depicted in the tops of the cones in Fig. 6.2.

**Lemma 6.1.1 (Covering Lemma).** Let $P = \{p_1, \ldots, p_n\}$ be a set of points ordered by a greedy permutation with insertion radii $\lambda_1, \ldots, \lambda_n$. For any $\alpha, \beta \geq 0$, and any $p_j \in P$, there exists a point $p_i \in P$ such that

1. if $\beta \geq \alpha$ then $b_j(\alpha) \subseteq b_i(\beta)$, and
2. if $\beta \geq (1 + \varepsilon)\alpha$, then $B(p_j, \alpha) \subseteq b_i(\beta)$.

**Proof.** Fix any $p_j \in P$. We may assume that $\beta \geq \lambda_j(1+\varepsilon)^2/\varepsilon$, for otherwise, choosing
\( p_i = p_j \) suffices to satisfy both clauses, the first because \( b_j(\alpha) \subseteq b_j(\beta) \) and the second because \( B(p_j, \alpha) = b_j(\alpha) \subseteq b_j(\beta) \). This assumption is equivalent to the assumption that \( b_j(\beta) = \emptyset \).

By the covering property of the greedy permutation, there is a point \( p_i \in P \) such that \( d(p_i, p_j) \leq \varepsilon \beta / (1 + \varepsilon) \) and \( \lambda_i \geq \varepsilon \beta / (1 + \varepsilon) \). It follows that \( r_i(\beta) = \beta \) and \( b_i(\beta) = B(p_i, \beta) \). Recall that \( \lambda_1 = \infty \) by convention, so \( b_1(\beta) \neq \emptyset \), and for large values of \( \beta \), choosing \( p_i = p_1 \) suffices.

To prove the first clause, fix any point \( x \in b_j(\alpha) \). By the triangle inequality,

\[
\begin{align*}
d(x, p_i) & \leq d(x, p_j) + d(p_i, p_j) \\
& \leq r_j(\alpha) + \frac{\varepsilon \beta}{1 + \varepsilon} \\
& \leq \frac{\lambda_j(1 + \varepsilon)}{\varepsilon} + \frac{\varepsilon \beta}{1 + \varepsilon} \quad \text{[we showed that} \beta \geq \frac{\lambda_j(1 + \varepsilon)^2}{\varepsilon}] \\
& \leq \beta = r_i(\beta).
\end{align*}
\]

So, \( x \in b_i(\beta) \) and thus, \( b_j(\alpha) \subseteq b_i(\beta) \) as desired.

To prove the second clause of the lemma, fix any \( x \in B(p_j, \alpha) \). By the triangle inequality,

\[
\begin{align*}
d(x, p_i) & \leq d(x, p_j) + d(p_i, p_j) \\
& \leq \alpha + \frac{\varepsilon \beta}{1 + \varepsilon} \\
& \leq \frac{\beta}{1 + \varepsilon} + \frac{\varepsilon \beta}{1 + \varepsilon} \quad \text{[}\beta \geq (1 + \varepsilon)\alpha\text{]} \\
& = r_i(\beta).
\end{align*}
\]

So, as before, \( x \in b_i(\beta) \) and thus, \( B(p_j, \alpha) \subseteq b_i(\beta) \) as desired.

**Corollary 6.1.2.** Let \( P = \{p_1, \ldots, p_n\} \) be a set of points ordered by a greedy permu-
tation with insertion radii $\lambda_1, \ldots, \lambda_n$. For all $\alpha \geq 0$,

$$\tilde{P}^\alpha = \bigcup_i b_i(\alpha)$$

and

$$\tilde{P}^\alpha \subseteq P^\alpha \subseteq \tilde{P}^{(1+\varepsilon)\alpha}.$$  

Proof. We will first show that $\tilde{P}^\alpha = \bigcup_i b_i(\alpha)$.

Fix any $\alpha \geq 0$. For all $j \in [n]$, $b_j(\alpha) \subseteq B(p_j, r_j(\alpha))$, so

$$\bigcup_{j \in [n]} b_j(\alpha) \subseteq \bigcup_{j \in [n]} B(p_j, r_j(\alpha)) = \tilde{P}^\alpha. \quad (6.1)$$

To show that $\tilde{P} = B(p_j, r_j(\alpha)) \subseteq \bigcup_{j \in [n]} b_j(\alpha)$, we have two cases. If $\alpha \leq \frac{\lambda_j(1+\varepsilon)^2}{\varepsilon}$, then $b_j(\alpha) = B(p_j, r_j(\alpha))$. Else $\alpha > \frac{\lambda_j(1+\varepsilon)^2}{\varepsilon}$, which implies that $r_j(\alpha) = \frac{\lambda_j(1+\varepsilon)}{\varepsilon}$. Let $\gamma = r_j(\alpha)$, which implies $r_j(\gamma) = \gamma$ and $\alpha > (1+\varepsilon)\gamma$, so there exists $i$ such that $B(p_j, \gamma) \subseteq b_i(\alpha)$ and equivalently $B(p_j, r_j(\alpha)) \subseteq b_i(\alpha)$. Thus,

$$\tilde{P} = \bigcup_{j \in [n]} B(p_j, r_j(\alpha)) \subseteq \bigcup_{j \in [n]} b_j(\alpha). \quad (6.2)$$

So (6.1) and (6.2) imply that $\tilde{P}^\alpha = \bigcup_i b_i(\alpha)$.

Now, we will prove that $\tilde{P}^\alpha \subseteq P^\alpha \subseteq \tilde{P}^{(1+\varepsilon)\alpha}$.

$$\tilde{P} = \bigcup_{j \in [n]} B(p_j, r_j(\alpha)) \subseteq \bigcup_{j \in [n]} B(p_j, \alpha) = P^\alpha, \quad (6.3)$$

because $r_j(\alpha) \leq \alpha$. Let $\beta = (1+\varepsilon)\alpha$, then for all $j \in [n]$ there exists $i$ such that
$B(p_j, \alpha) \subseteq b_i(\beta)$ by statement 2 in Lemma 6.1.1, implying

$$P^\alpha = \bigcup_{j \in [n]} B(p_j, \alpha) \subseteq \bigcup_{j \in [n]} b_j(\beta) = \tilde{P}^{\beta} = \tilde{P}^{(1+\varepsilon)\alpha}.$$ 

(6.4)

Thus (6.3) and (6.4) imply that $\tilde{P}^\alpha \subseteq P^\alpha \subseteq \tilde{P}^{(1+\varepsilon)\alpha}$

Corollary 6.1.2 implies the following proposition using standard results on the stability of persistence barcodes [19].

**Proposition 6.1.3.** The persistence barcode of the perturbed offsets $\{\tilde{P}^\alpha\}_{\alpha \geq 0}$ is a $(1 + \varepsilon)$-approximation to the persistence barcode of the offsets $\{P^\alpha\}_{\alpha \geq 0}$.

### 6.2 Sparse Filtrations

The *sparse Čech complex* is defined as $Q^\alpha := \text{Nrv}\{b_i(\alpha) \mid i \in [n]\}$. Notice that because $b_i(\alpha) = \emptyset$ unless $\lambda_i$ is sufficiently large compared to $\alpha$, there are fewer vertices as the scale increases. This is the desired sparsification. Unfortunately, it means that the set of complexes $\{Q^\alpha\}$ is not a filtration, but this is easily remedied by the following definition. The *sparse Čech filtration* is defined as $\{S^\alpha\}$, where

$$S^\alpha := \bigcup_{\delta \leq \alpha} Q^\delta = \bigcup_{\delta \leq \alpha} \text{Nrv}\{b_i(\delta) \mid i \in [n]\}.$$

This definition makes it clear that the sparse complex is a union of nerves, but it not obvious that it has the same persistent homology as the filtration defined by the perturbed offsets $\tilde{P}^\alpha := \bigcup_i b_i(\alpha)$. For such a statement, it would be much more convenient if $\{S^\alpha\}$ was itself a nerve filtration rather than a union of nerves, in which case the Persistent Nerve Lemma could be applied directly. In fact, this can be done by adding an extra dimension corresponding to the filtration parameter extending the
balls $b_i(\alpha)$ into the perturbed cone shapes

$$U_i^\alpha := \bigcup_{\delta \leq \alpha} (b_i(\delta) \times \{\delta\}).$$

These sets, depicted in Figs. 6.2 and 6.3, allow the following equivalent definition of the complexes in the sparse Čech filtration.

$$S^\alpha := \text{Nrv}\{U_i^\alpha \mid i \in [n]\}.$$  

**Proposition 6.2.1.** If $d$ is a convex metric and $r_i$ is a concave function then $U_i^\alpha := \bigcup_{\delta \leq \alpha} (b_i(\delta) \times \{\delta\})$ is convex.

**Proof.** Given two points $(a, \delta_a), (b, \delta_b) \in U_i^\alpha$, $d(a, p_i) \leq r_i(\delta_a)$ and likewise $d(b, p_i) \leq r_i(\delta_b)$ by definition of $r_i$. Let $c = (1 - t)a + tb$ and let $\delta_c = (1 - t)\delta_a + t\delta_b$, for $t \in [0, 1]$.

Now we bound $d(c, p_i)$ as follows.

$$d(c, p_i) \leq (1 - t)d(a, p_i) + td(b, p_i) \quad [d \text{ is convex}]$$
$$\leq (1 - t)r_i(\delta_a) + tr_i(\delta_b)$$
$$\leq r_i(\delta_c) \quad [r_i \text{ is concave}]$$

Thus we can conclude that $(c, \delta_c)$, a convex combination of arbitrary $(a, \delta_a)$ and $(b, \delta_b)$, is in $U_i^\alpha$ and $U_i^\alpha$ is convex. \hfill \Box

**Theorem 6.2.2.** The persistence barcode of the sparse nerve filtration $\{S^\alpha\}_{\alpha \geq 0}$ is a $(1 + \varepsilon)$-approximation to the persistence barcode of the offsets $\{P^\alpha\}_{\alpha \geq 0}$.

**Proof.** For all $i$, the set $U_i^\alpha$ is convex because $r_i$ is concave by Proposition 6.2.1. It follows that the sets $U_i^\alpha$ satisfy the conditions of the Persistent Nerve Lemma. So, $\{S^\alpha\}$ has the same persistence barcode as the filtration $\{B^\alpha\}$, where $B^\alpha := \bigcup_i U_i^\alpha$.  

115
The Covering Lemma implies that the linear projection of $B^\alpha$ to $\tilde{P}^\alpha$ that maps $(x, \delta)$ to $x$ is a homotopy equivalence as each fiber is simply connected. Moreover, the projection clearly commutes with the inclusions $B^\alpha \hookrightarrow B^\beta$ and $\tilde{P}^\alpha \hookrightarrow \tilde{P}^\beta$, from which, it follows that $\text{Pers}\{\tilde{P}^\alpha\} = \text{Pers}\{B^\alpha\} = \text{Pers}\{S^\alpha\}$. So, the claim now follows from Proposition 6.1.3.

\[\square\]

6.3 Algorithms to Construct Sparse Filtrations

In previous work, it was shown how to use metric data structures \[41\] to compute the sparse Rips filtration in $O(n \log n)$ time \[63\] when the doubling dimension is constant. The same approach also works for the sparse nerve filtrations described here. However, it depends on the construction of a net-tree \[41\], which is an intricate data structure.

In this section, we present a simpler technique to construct a sparse nerve filtration from a greedy permutation of a finite point set $P$ in metric $(\mathcal{M}, d)$. Throughout, we assume that the doubling dimension of $P$ is constant. We show how to construct a sparse nerve filtration in linear time from the greedy permutation. Our approach starts with finding all edges and their birth times.

Let $G$ be a directed graph called the neighborhood graph whose vertices are the points of $P$ and whose edges are the edges of the sparse nerve filtration of $P$ directed
from smaller to larger insertion radius. In Section 6.3.1, it is shown that for each directed edge \((p_i, p_j)\) in \(G\), \(d(p_i, p_j) \leq \kappa \lambda_i\), for a constant \(\kappa\). This reduces the problem of finding the edges of the filtration to the problem of finding points in a given neighborhood. Moreover, we show that the out-degree of a vertex in \(G\) is constant. Then, in Section 6.3.2, we present an algorithm to construct \(G\) from the greedy permutation and show that it runs in linear time. Finally, in Section 6.3.3, we give an algorithm for building higher dimensional simplices using the directed graph and bound its running time.

### 6.3.1 Finding Neighborhoods Suffices

The vertices adjacent to \(p_i\) in the directed graph \(G\) are the points \(p_j\) with insertion radius at least that of \(p_i\) such that their corresponding balls intersect at some scale \(\alpha\). The following lemma shows that these points have distance at most a constant times \(\lambda_i\) to \(p_i\). Then, Lemma 6.3.2 will use this fact to show that the number of adjacent vertices is at most a constant.

**Lemma 6.3.1.** For a given point \(p_i\) with insertion radius \(\lambda_i\) in the directed graph \(G\), all adjacent points to \(p_i\) are located in a \(B(p_i, \kappa \lambda_i)\), where \(\kappa = \frac{\varepsilon^2 + 3\varepsilon + 2}{\varepsilon}\) and \(\varepsilon > 0\).

**Proof.** In the directed graph \(G\), a vertex \(p_j\) is adjacent to vertex \(p_i\) if \(\lambda_i \leq \lambda_j\) and for some scale \(\alpha\), \(b_i(\alpha) \cap b_j(\alpha) \neq \emptyset\). These balls intersect before \(p_i\) disappears, so

\[
b_i\left(\frac{\lambda_i(1 + \varepsilon)^2}{\varepsilon}\right) \cap b_j\left(\frac{\lambda_j(1 + \varepsilon)^2}{\varepsilon}\right) \neq \emptyset.
\]
The distance between $p_i$ and $p_j$ is bounded as follows.

\[
\begin{align*}
d(p_i, p_j) & \leq r_i \left( \frac{\lambda_i(1+\varepsilon)^2}{\varepsilon} \right) + r_j \left( \frac{\lambda_j(1+\varepsilon)^2}{\varepsilon} \right) \quad \text{[triangle inequality]} \\
& \leq \frac{\lambda_i(1+\varepsilon)}{\varepsilon} + r_j \left( \frac{\lambda_i(1+\varepsilon)^2}{\varepsilon} \right) \quad \text{[for } \alpha > \frac{\lambda_i(1+\varepsilon)}{\varepsilon}, r_i(\alpha) = \frac{\lambda_i(1+\varepsilon)}{\varepsilon}]} \\
& \leq \frac{\lambda_i(1+\varepsilon)}{\varepsilon} + \frac{\lambda_j(1+\varepsilon)^2}{\varepsilon} \quad \text{[for } \alpha < \frac{\lambda_j(1+\varepsilon)}{\varepsilon}, r_j(\alpha) = \alpha]} \\
& \leq \frac{\varepsilon^2 + 3\varepsilon + 2}{\varepsilon} \lambda_i.
\end{align*}
\]

Thus, all adjacent vertices to $p_i$ lie in a ball with center $p_i$ and radius $\kappa \lambda_i$. \qed

**Lemma 6.3.2.** For a point set $P$ ordered by a greedy permutation and with doubling constant $\rho$, each $p_i \in P$ has $\rho O(\log \kappa)$ neighbors in the directed graph $G$, where $\kappa = \frac{\varepsilon^2 + 3\varepsilon + 2}{\varepsilon}$ and $\varepsilon > 0$.

**Proof.** Using Lemma 6.3.1, neighbors of $p_i$ are in $B(p_i, \kappa \lambda_i)$ and they are pairwise $\lambda_i$-separated. Therefore, the Packing Lemma (Lemma 2.1.1) implies that $p_i$ has at most $\rho O(\log \kappa)$ neighbors. \qed

### 6.3.2 Constructing the Neighborhood Graph

In this section, we construct the directed graph $G$ as described from a given greedy permutation. In Section 6.3.1, it was shown that to construct $G$ it suffices to find points within a metric ball around each point. We build an efficient data structure to maintain these points. Our approach is similar to the incremental algorithm for constructing a net-tree from a greedy permutation [41], but it is simpler because it does not construct the whole net-tree and stores only one node per input point. This data structure for enumerating neighborhoods may be of independent interest.

Let $P = \{p_1, \ldots, p_n\}$ be the points in $(\mathcal{M}, d)$ ordered according to a greedy permutation. For each $p_i \in P$, let $\text{pred}(p_i) \in P_{i-1}$ denote the nearest point to $p_i$ among the
first \( i - 1 \) points in the ordering. So, the insertion radius of \( p_i \) is \( \lambda_i := d(p_i, \text{pred}(p_i)) \). The \emph{level} of \( p_i \) is defined as \( \ell_i := \lfloor \lg \lambda_i \rfloor \).

The goal is to process the points one at a time in the greedy ordering, and for each \( p_i \), to find all preceding points within distance \( \kappa \lambda_i \), where \( \kappa = (\varepsilon^2 + 3\varepsilon + 2)/\varepsilon \) and \( \varepsilon > 0 \) is a fixed constant chosen by the user. Because all neighbors of \( p_i \) in a sparse nerve filtration have this property by Lemma 6.3.1, we can use this list to find all the neighbors.

We will define a data structure \( D \) used to extract neighborhood information in the directed graph \( G \). For each point \( p_i \) in \( P \), \( D \) stores \( \text{pred}(p_i) \), \( \ell_i \), and three other pieces of information:

1. a point \( \text{par}(p_i) \) called the \emph{parent},
2. a list of points \( \text{nbr}(p_i) \) called the \emph{neighbors}, and
3. a list of points \( \text{ch}(p_i) \) called the \emph{children} of \( p_i \).

These three objects change over the course of the algorithm. We only require that for all \( i \in [n] \) and all \( p_j \in P_i \), they satisfy the following invariants after \( i \) points have been processed.

1. **Parent Invariant:** \( \text{par}(p_j) = p_j \) if \( \ell_j > \ell_i \). Otherwise, \( \text{par}(p_j) \) is a point \( p_k \) such that \( \ell_k > \ell_i \) and \( d(p_j, p_k) \leq 2^\ell_i \).
2. **Child Invariant:** \( \text{ch}(p_j) \supseteq \{p_j\} \cup \{p_k \in P_i \mid \text{par}(p_k) = p_j \text{ and } \ell_k = \ell_i\} \).
3. **Neighbor Invariant:** \( \text{nbr}(p_j) \supseteq \{p_k \in P_i \mid d(p_j, p_k) \leq \kappa 2^{\min(\ell_j, \ell_k, \ell_i + 1)}\} \).

The second invariant states that the children list of \( p_j \) contains all points at the same level as \( p_i \) that have \( p_j \) as a parent. The third invariant says that the neighbor lists contain all nearby points where “nearby” is related to the insertion radius of \( p_i \). This
last invariant implies the correctness of the algorithm, because for \( j = i \), it says the neighbor list contains the set we are interested in. We maintain the lists for the other points to help us do updates at each step.

Furthermore, we assume that \( \mathcal{D} \) provides constant-time access to the list of points in a specific level.

Algorithm 11 shows how a new point \( p_i \) can be inserted into the data structure \( \mathcal{D} \). In fact, we process points of a greedy permutation one by one and after inserting a new point in \( \mathcal{D} \), we update the directed graph \( G \), which is used to extract higher dimensional simplices.

**Algorithm 11** Inserting a new point into the data structure \( \mathcal{D} \)

1: **procedure** \( \text{Insert}(\mathcal{D}, p_i) \)
2: \[ \text{if } \ell_i < \ell_{i-1} \text{ then} \]
3: \[ \text{for all } p_k \text{ such that } \ell_k = \ell_{i-1} \text{ do} \]
4: \[ \text{par}(p_k) \leftarrow p_k \]
5: \[ p_j \leftarrow \text{pred}(p_i) \]
6: \[ \text{par}(p_i) \leftarrow \text{par}(p_j) \]
7: \[ \text{for all } p_k \in \text{nbr}(\text{par}(p_j)) \text{ do} \]
8: \[ \text{if } d(p_i, p_k) \leq d(p_i, \text{par}(p_i)) \text{ and } \ell_k > \ell_i \text{ then} \]
9: \[ \text{par}(p_i) \leftarrow p_k \]
10: \[ \text{add } p_i \text{ to } \text{ch}(p_i) \]
11: \[ \text{add } p_i \text{ to } \text{ch}(\text{par}(p_i)) \]
12: \[ \text{add } p_i \text{ to } \text{nbr}(p_i) \]
13: \[ \text{for all } p_k \in \text{ch}(\text{nbr}(\text{par}(p_i))) \text{ do} \]
14: \[ \text{if } d(p_i, p_k) \leq \kappa 2^\ell_i \text{ then} \]
15: \[ \text{add } p_k \text{ to } \text{nbr}(p_i) \]
16: \[ \text{add } p_i \text{ to } \text{nbr}(p_k) \]

**Lemma 6.3.3.** Let \( P = (p_1, \ldots, p_n) \) be a greedy permutation. For all \( i \in \{2, \ldots, n\} \),
if $D$ is a data structure on $P_{i-1}$ satisfying the three invariants, then it also satisfies the invariants after calling $\text{INSERT}(D, p_i)$.

**Proof.** We consider the invariants one at a time.

First, if $\ell_i < \ell_{i-1}$, the algorithm updates the parents of all nodes in level $\ell_{i-1}$. Note that these are the only points required to be updated to satisfy the Parent Invariant for all points in $P_{i-1}$.

Next, we check that there exists a point $p_k$ such that setting $\text{par}(p_i)$ to $p_k$ satisfies the Parent Invariant. The algorithm iterates over $\text{nbr}(\text{par}(p_j))$ to find the closest point with a level higher than $\ell_i$. We first show there exists a point in a higher level that satisfies the Parent Invariant and then show that any such point is in $\text{nbr}(\text{par}(p_j))$. Let $z := \arg \max_{z<i} \{ \ell_z | \ell_z > \ell_i \}$. Let $p_k$ be the closest point in $P_z$ to $p_i$. So,

$$d(p_i, p_k) = d(p_i, P_z) \leq \max_{p \in P} d(p, P_z) = \lambda_{z+1} \leq 2^{\ell_{z+1}} \leq 2^{\ell_i}.$$ 

Thus, some point $p_k$ could satisfy the Parent Invariant. Any such point $p_k$ satisfies

$$d(p_k, \text{par}(p_j)) \leq d(p_k, p_i) + d(p_i, p_j) + d(p_j, \text{par}(p_j)) \quad \text{[triangle inequality]}$$

$$\leq 2^{\ell_i} + \lambda_i + 2^{\ell_i} \quad \text{[Parent inv., $p_j = \text{pred}(p_i)$]}$$

$$< 2^{\ell_i} + 2^{\ell_i} + 2^{\ell_i} \quad \text{[$\ell_i = \lceil \lambda_i \rceil$]}$$

$$= \frac{3}{2} \cdot 2^{\ell_i+1}$$

$$< \kappa 2^{\ell_i+1}.$$ 

Therefore, $p_k \in \text{nbr}(\text{par}(p_j))$ by the Neighbor Invariant.

For the Child Invariant, $p_i$ needs to be inserted into $\text{ch}(\text{par}(p_i))$. No other children lists need to change to satisfy the invariant.

Next, to satisfy the Neighbor Invariant, neighbor lists should be updated. This
only involves finding the neighbor list of \( p_i \) and also adding \( p_i \) to the neighbor lists of its neighbors. For this step, it suffices to check that if \( p_k \) must be added to \( \text{nbr}(p_i) \), i.e. if \( d(p_i, p_k) \leq \kappa 2^{\ell_i} \), then \( p_k \in \text{ch(nbr(par(p_i)))} \). That is, the neighbors of \( p_i \) are all children of neighbors of the parent of \( p_i \). This follows from the triangle inequality and the invariants for \( i - 1 \) as follows.

\[
d(\text{par}(p_k), \text{par}(p_i)) \leq d(\text{par}(p_k), p_k) + d(p_k, p_i) + d(p_i, \text{par}(p_i)) \\
\leq 2^{\ell_i} + \kappa 2^{\ell_i} + 2^{\ell_i} \\
= (1 + \kappa/2)2^{\ell_i+1} \\
< \kappa 2^{\ell_i+1}. \quad [\kappa > 2]
\]

So, it follows that \( \text{par}(p_k) \in \text{nbr(par}(p_i)) \), and so \( p_k \in \text{ch(par}(p_k)) \subseteq \text{ch(nbr(par}(p_i))) \). If \( p_k \) is added to \( \text{nbr}(p_i) \), then it is required to add \( p_i \) to \( \text{nbr}(p_k) \) and the algorithm does this.

Algorithm 12 constructs all edges that appear in a sparse filtration. It receives a set of points \( P \), which is ordered by a greedy permutation, as input and returns a directed graph \( G \). As we mentioned earlier, we will use the directed graph \( G \) to find higher dimensional simplices. For each point \( p_i \), the algorithm invokes the \text{INSERT} procedure to find its neighbors. Then, to build sparse edges between \( p_i \) and its neighbors, Algorithm 13 is called. If an edge appears in the sparse filtration, \text{EDGE\text{\textsc{Birth\textsc{Time}}} method returns the birth time of the edge and } \infty \text{ otherwise. Finally, for an edge in the sparse filtration, a directed edge from } p_i \text{ to } p_j \text{ will be inserted into } G.

**Theorem 6.3.4.** Given a greedy permutation of a point set \( P \) with the doubling constant \( \rho > 0 \) and the nearest predecessors \( \text{pred}(p) \) for each \( p \in P \), one can compute the edges of the sparse nerve filtration \( \{S^\alpha\}_{\alpha \geq 0} \) of \( P \) in \( O(\rho O^{(\log \kappa)} n) \) time for a constant
Algorithm 12 Constructing edges of a sparse filtration

1: procedure \textsc{ConstructEdges}(P = \{p_1, \ldots, p_n\})
2: \hspace{1em} Initialize \mathcal{D} with p_1 \quad \triangleright \text{ adds } p_1 \text{ to } \text{ch}(p_1) \text{ and sets } \text{par}(p_1) = p_1.
3: \hspace{1em} Initialize a directed graph \(G\) on \(P\)
4: \hspace{1em} for \(i = 2\) to \(n\) do
5: \hspace{2em} \textsc{Insert}(\mathcal{D}, p_i)
6: \hspace{2em} for all \(p_j \in \text{nbr}(p_i)\) do
7: \hspace{3em} \(\alpha \leftarrow \text{EdgeBirthTime}(p_i, p_j)\)
8: \hspace{3em} if \(\alpha < \infty\) then
9: \hspace{4em} \text{Add a directed edge from } p_i \text{ to } p_j \text{ with birth time } \alpha \text{ to } G
10: \hspace{1em} return \(G\)

\(\varepsilon > 0\) and \(\kappa = \frac{\varepsilon^2 + 3\varepsilon + 2}{\varepsilon}\).

Proof. Algorithm 12 finds all edges in a sparse filtration. The running time of this algorithm mainly depends on the running time of \textsc{Insert} procedure and the size of neighbor list for each point.

In Algorithm 11, the most common operation for the lists \(\text{nbr}(p_j)\) and \(\text{ch}(p_j)\) is to enumerate their elements. Any time a list is enumerated, we can check each point in constant time to see if it is still required to satisfy the invariant and remove it otherwise. Note that although the invariants only specify a subset that must appear, it is easy to check that enumerating these lists can be done in amortized constant time. This follows from two facts. First, the required subsets have constant size (by the Packing Lemma). Second, the number of removals is at most the number of insertions, so we charge the cost of visiting such a point in the enumeration to the cost of its insertion.

In addition, when inserting \(p_i\), if \(\ell_i < \ell_{i-1}\), then \(\text{par}(p_k)\) is updated for all \(p_k\) such that \(\ell_k = \ell_{i-1}\). The total cost of such operations is \(O(n)\) as no parent is updated twice.

After insertion of a point \(p_i\) into \(\mathcal{D}\), Algorithm 13 is called for all points in \(\text{nbr}(p_i)\) to check whether an edge belongs to the sparse filtration. This algorithm has a constant
Algorithm 13 Compute the birth time of an edge

1: procedure EdgeBirthTime($p_i, p_j$)
2:     if $\lambda_i > \lambda_j$ then
3:         swap $p_i$ and $p_j$
4:     if $d(p_i, p_j) \leq \frac{2\lambda_i(1+\varepsilon)}{\varepsilon}$ then
5:         return $\frac{d(p_i, p_j)}{2}$
6:     if $d(p_i, p_j) \leq \frac{(\lambda_i+\lambda_j)(1+\varepsilon)}{\varepsilon}$ then
7:         return $d(p_i, p_j) - \frac{\lambda_i(1+\varepsilon)}{\varepsilon}$
8:     return $\infty$

running time. In addition, by Lemma 6.3.2, the size of a neighbor list for each point is constant. Therefore, for each point, the cost of finding these edges in the sparse filtration in $O(\rho^{O(\log \kappa)})$.

6.3.3 Higher Dimensional Simplices

In the previous section, it is shown that from a greedy permutation, the edges of a sparse nerve filtration can be constructed in linear time. Now, we present an algorithm to find $k$-simplices in the sparse filtration for $k > 1$. As mentioned earlier, the directed graph $G$ built from the edges of the sparse nerve filtration will be used to construct higher dimensional simplices.

Let $E(v)$ be the vertices adjacent to a vertex $v$ in $G$ (for each $u \in E(v)$, there is a directed edge from $v$ to $u$). To find a $k$-simplex for $k > 1$ containing a vertex $v$, we consider all subsets $\{u_1, \ldots, u_k\}$ of $k$ vertices in $E(v)$. If $\{v, u_1, \ldots, u_k\}$ forms a $(k+1)$-clique, we check the clique to see whether it creates a $k$-simplex and compute
its birth time. The birth time of a $k$-simplex $\sigma$ in a nerve filtration is defined as follows.

$$\text{SimplexBirthTime}(\sigma) := \min \left\{ \alpha : \bigcap_{j \in \sigma} U_j^\alpha \neq \emptyset \right\} = \min \left\{ \alpha : \bigcap_{j \in \sigma} b_j(\alpha) \neq \emptyset \right\}.$$ 

If no such $\alpha$ exists, then we define the birth time to be $\infty$. We assume the user provides a method, SimplexBirthTime, to compute birth times for their metric that runs in time polynomial in $k$. This function takes a $(k+1)$-clique as input. If at some scale $\alpha$, the corresponding balls have a common intersection, it returns the minimum such $\alpha$, otherwise, it returns $\infty$ indicating the $(k+1)$-clique is not a $k$-simplex in the sparse filtration.

For the case of Rips filtrations (i.e. $\ell_\infty$), SimplexBirthTime($\sigma$) just needs to compute the maximum birth time of the edges and compare it to $\min_{p_i \in \sigma} \lambda_i (1 + \varepsilon)^2 / \varepsilon$ (the first time $t$ after which some $p_i \in \sigma$ has $b_i(t) = \emptyset$). For $\ell_2$, the corresponding computation is a variation of the minimum enclosing ball problem.

Algorithm 14 finds the $k$-simplices and birth times in a sparse filtration. In this algorithm, $G$ is the given directed graph and the output $S$ is the set of pairs $(\sigma, t)$, where $\sigma$ is a $k$-simplex and $t$ is its birth time.

**Theorem 6.3.5.** Given the edges of a sparse nerve filtration, Algorithm 14 finds the $k$-simplices of $\{S^\alpha\}_{\alpha \geq 0}$ in $O(\rho^O(k \log \kappa) n)$ time, where $\rho > 0$ is the doubling constant, $\kappa = \frac{\varepsilon^2 + 3\varepsilon + 2}{\varepsilon}$, and $\varepsilon > 0$.

**Proof.** In Algorithm 14, for every vertex $v$ in the directed graph $G$, there are $\binom{|E(v)|}{k}$ subsets with size $k$. In addition, by Lemma 6.3.2, $|E(p_i)| = \rho^O(\log \kappa)$. Therefore, the total running time of this algorithm will be $\kappa^O(k \rho) n$. 

\[\square\]
Algorithm 14 Find all $k$-simplices and birth times

1: **procedure** FindSimplices($G, k$)
2: \hspace{1em} $S \leftarrow \emptyset$
3: \hspace{1em} **for all** vertex $v$ in $G$ **do**
4: \hspace{2em} **for all** $\{u_1, \ldots, u_k\} \subseteq E(v)$ **do**
5: \hspace{3em} if $\{v, u_1, \ldots, u_k\}$ is a $(k+1)$-clique **then**
6: \hspace{4em} $\sigma \leftarrow \{v, u_1, \ldots, u_k\}$
7: \hspace{4em} $t \leftarrow \text{SimplexBirthTime}(\sigma)$
8: \hspace{3em} if $t < \infty$ **then**
9: \hspace{4em} $S \leftarrow S \cup (\sigma, t)$
10: \hspace{1em} **return** $S$

6.4 Removing Vertices

Because the sparse filtration is a true filtration, no vertices are removed. When the cone is truncated, no new simplices will be added using that vertex, but it is still technically part of the filtration. The linear-size guarantee is a bound on the total number of simplices in the complex. Thus, by using methods such as zigzag persistence or simplicial map persistence to fully remove these vertices when they are no longer needed cannot improve the asymptotic performance. Still, it may be practical to remove them (see [11]).

In this section, we show that the geometric construction leads to a natural choice of elementary simplicial maps (edge contractions) which all satisfy the so-called link condition. In the persistence by simplicial maps work of Dey et al. [28] and Boissonat et al. [10], a key step in updating the data structures to contract an edge is to first add simplices so that the so-called Link Condition is satisfied. The *link* of a simplex $\sigma$ in
a complex \( K \) is defined as

\[
\text{Lk} \sigma = \{ \tau \setminus \sigma \mid \tau \in K \text{ and } \sigma \subseteq \tau \}.
\]

That is, the link \( \sigma \) is formed by removing the vertices of \( \sigma \) from each of its cofaces. An edge \( \{u, v\} \in K \) satisfies the \textit{Link Condition} if and only if

\[
\text{Lk} \{u, v\} = \text{Lk} \{u\} \cap \text{Lk} \{v\}.
\]

Dey et al. [27] proved that edge contractions induce homotopy equivalences when the link condition is satisfied. Thus, it gives a minimal local condition to guarantee that the contraction preserves the topology. More recently, it was shown that such a contraction does not change the persistent homology [28].

**Proposition 6.4.1.** If \((P, d)\) is a finite subset of a convex metric space and \(\{S^\alpha\}\) is its corresponding sparse filtration, then the last vertex \(p_n\) has a neighbor \(p_i\) such that the edge \(\{p_n, p_i\} \in S^\alpha\) satisfies the link condition, where \(\alpha = \lambda_n(1 + \varepsilon)^2/\varepsilon\) and \(\lambda_n\) is the insertion radius of \(p_n\).

**Proof.** It follows directly from the definition of a link that \(\text{Lk} \{u, v\} \subseteq \text{Lk} \{u\} \cap \text{Lk} \{v\}\) for all edges \(\{u, v\}\). By the Covering Lemma (Lemma 6.1.1), we know that there exists a \(p_i \in P\) such that \(b_n(\alpha) \subseteq b_i(\alpha)\). Thus, it suffices to check that \(\text{Lk} \{i\} \cap \text{Lk} \{n\} \subseteq \text{Lk} \{i, n\}\). Because the vertices are ordered according to a greedy permutation, \(\lambda_n \geq \lambda_j\) for all \(p_j \in P\). It follows that a simplex \(J \in S^\alpha\) if and only if \(\bigcap_{i \in J} b_j(\alpha) \neq \emptyset\).

Let \(J\) be any simplex in \(\text{Lk} \{i\} \cap \text{Lk} \{n\}\). So, \(i, n \notin J\) and \(\bigcap_{j \in J \cup \{n\}} b_j(\alpha) \neq \emptyset\). Because \(b_n(\alpha) \cap b_i(\alpha) = b_n(\alpha)\), it follows that \(\bigcap_{j \in J \cup \{i, n\}} b_j(\alpha) \neq \emptyset\). Thus, we have \(J \in \text{Lk} \{i, n\}\) as desired. \(\square\)
Chapter 7

Conclusion and Future Work

In this dissertation, we proposed a hierarchical data structure to organize data residing in doubling metric spaces. We proved that our data structure satisfies the global conditions of the net-tree, a theoretically optimal structure that solves many geometric problems efficiently. We called our data structure local net-trees and presented algorithms to construct them. Our randomized incremental construction algorithm runs in $O(n \log n)$ expected time matching the best theoretical bound. In addition, it is much simpler than previous work [41, 24] and uses smaller constants. We implemented our algorithms in Python and ran experiments to observe their performance in practice. In our experiments, we observed that $\tau = 6$ is a good choice for the scale factor. In addition, we observed that the running time of our randomized incremental algorithm has a direct relationship with the fraction of basic touches required in the point location step.

Along the way, we proposed a linear time algorithm to convert a cover tree into

This research was sponsored by the National Science Foundation under grant numbers CCF-1525978 and CCF-1652218. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.
a net-tree. Cover trees are the simplest data structures for general metric spaces. Although they are not theoretically optimal, they work well in practice and have been used extensively for machine learning applications. We also presented two linear time operations to make a net-tree finer or coarser. These two operations help us to make a trade-off between the height and the number of edges of a net-tree. Also, in Chapter 5, we discussed how the refining operation can be used to decrease the aspect ratio of a locally greedy permutation constructed from a net-tree.

We generalized the definition of greedy permutations in Chapter 5 and introduced two wider class of permutations called approximate greedy and locally greedy. In addition, we showed that there is a close relationship between these permutations and net-trees. We also presented linear time algorithms to go back and forth between net-trees and these permutations.

In Chapter 6, we showed how hierarchical metric trees can be applied to a topological data analysis problem. We gave a new geometric perspective on sparse filtrations for topological data analysis that leads to a simple proof of correctness for all convex metrics. By considering a nerve construction one dimension higher, the proofs are primarily geometric and do not require explicit construction of simplicial maps. This geometric view clarifies the non-zigzag construction, while also showing that removing vertices can be accomplished with simple edge contractions. We presented a linear time algorithm to create a sparse filtration from a greedy permutation of points in a doubling metric.

As future work, we plan to prove that the lower bound of $\Omega(n \log n)$ holds for the greedy permutation problem. Also, we plan to devise algorithms to make a trade-off between the construction time and the aspect ratio of locally greedy permutations. Another interesting problem is designing an efficient algorithm to find all point predecessors in a greedy permutation. Furthermore, we plan to modify our sparse filtration
analyses and algorithms for approximate greedy and locally greedy permutations.
Bibliography


