Algorithms for Analyzing Spatio-Temporal Data

by

Abhinandan Nath

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Pankaj K. Agarwal, Supervisor

_____
Kamesh Munagala

_____
Rong Ge

_____
Yusu Wang

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2018

Abstract

Algorithms for Analyzing Spatio-Temporal Data

by

Abhinandan Nath

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Pankaj K. Agarwal, Supervisor

_____
Kamesh Munagala

_____
Rong Ge

_____
Yusu Wang

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2018

Abstract

In today's age, huge data sets are becoming ubiquitous. In addition to their size, most of these data sets are often noisy, have outliers, and are incomplete. Hence, analyzing such data is challenging. We look at applying geometric techniques to tackle some of these challenges, with an emphasis on designing provably efficient algorithms. Our work takes two broad approaches – distributed algorithms, and concise descriptors for big data sets.

With the massive amounts of data available today, it is common to store and process data using multiple machines. Parallel programming frameworks such as MapReduce and its variants are becoming popular for handling such large data. We present the first provably efficient algorithms to compute, store, and query data structures for range queries and approximate nearest neighbor queries in a popular parallel computing abstraction that captures the salient features of MapReduce and other massively parallel communication (MPC) models. Our algorithms are competitive in terms of running time and workload to their classical counterparts.

We propose parallel algorithms in the MPC model for processing large terrain elevation data (represented as a 3D point cloud) that are too big to fit on one machine. In particular, we present a simple randomized algorithm to compute the Delaunay triangulation of the xy-projections of the input points. Next we describe an efficient algorithm to compute the contour tree (a topological descriptor that succinctly encodes the contours of a terrain) of the resulting triangulated terrain.

We then look at comparing real-valued functions, by computing a distance function between their merge trees (a small-sized descriptor that succinctly captures the sublevel sets of a function). Merge trees are robust to noise in the data, and can be used effectively as proxies for the data sets themselves in some cases. We also use it give an algorithm to compute the Gromov-Hausdorff distance, a natural way to measure distance between two metric spaces. We give the first proof of hardness and the first non-trivial approximation algorithm for computing the Gromov-Hausdorff distance between metric spaces defined on trees, where the distance between two points is given by the length of the unique path between them in the tree.

Finally we look at the problem of capturing shared portions between large number of input trajectories. We formulate it as a subtrajectory clustering problem - the clustering of subsequences of trajectories. We propose a new model for clustering subtrajectories. Each cluster of subtrajectories is represented as a *pathlet*, a sequence of points that is not necessarily a subsequence of an input trajectory. We present a single objective function for finding the optimal collection of pathlets that best represents the trajectories taking into account noise and other artifacts of the data. We show that the subtrajectory clustering problem is NP-hard and present fast approximation algorithms. We further improve the running time of our algorithm if the input trajectories are "well-behaved". We also present experimental results on both real and synthetic data sets.

Dedicated to everyone who inspired me.

Contents

List of Tables

List of Figures

Acknowledgements

This dissertation would not have been possible without help from a lot of people.

First and foremost, I would like to thank my advisor Pankaj. More than anyone else, he has taught me what research is – how to read a paper, how to choose a problem to work on, and how to present my ideas to other people. Perhaps the most important lesson that I learned from him is that it is more important to properly formulate a problem than to actually solve it.

I would like to thank Kamesh for serving on my committee. He has been a long-time collaborator and a source of advice in our numerous meetings. Both he and Pankaj were extemely patient in answering all of my questions, and slowing down their thought processes so that I could catch up.

Along with Pankaj, Kyle has been a constant collaborator on all my papers. It has been nothing short of inspiring to see his enthusiasm, work ethic, and dedication. I would also like to thank the other members of my committee – Rong and Yusu – for helpful discussions, comments and feedback. Thanks are also due to my other wonderful collaborators – Yusu, Tasos, Jiangwei, and Erin.

Thanks to my fellow students – Aaron, Abhishek, Allen, Alex, Ergys, Ios, Jiangwei, Mayuresh, Stavros, Wuzhou – for making my time so fun and enjoyable. The awesome staff members at Duke CS – Marilyn, Pam, Alison, Kathleen, Joe, Jeff – made sure that we could focus entirely on our research without having to worry about anything else. I feel a deep sense of gratitude towards them. My friends outside of

work – Aditi, Harish, Tushar, Seth, Preetish, members of my volleyball cohort – made sure that my sanity was maintained throughout these years.

Thanks to the Core Ranking team at A9, Palo Alto, especially my manager Chris Severs and my mentor Nick Blumm, where I spent the summer of 2017 as an intern.

Thanks to the various funding sources that supported me over the past few years – Duke Computer Science, Duke Graduate School, National Science Foundation, Army Research Office, and the US-Israeli Binational Science Foundation.

Last but not the least, I would like to thank my family – my parents and my younger brother – for all the sacrifices, and for always being there when I needed them the most.

1

Introduction

We are drowning in a world of data. To quote an article that appeared in 2014 [1] –

"By 2020 the digital universe will contain nearly as many digital bits as there are stars in the universe. It is doubling in size every two years, and by 2020 the digital universe – the data we create and copy annually – will reach 44 trillion gigabytes".

Data is being collected, knowingly or unknowingly, scrupulously or unscrupulously, from almost everywhere and all the time. Our cellphones are continously sending location and other information to servers; websites track our activity constantly; sensors placed on the ocean floor are collecting and sending seismological data for analysis purposes [2]; weather satellites are continuously taking pressure and humidity measurements, and beaming them to weather stations for forecasting. The ethical and moral issues concerning data privacy and data abuse pose very important and interesting questions, but these are outside the purview of this dissertation.

Many of these data sets have a geometric flavor. For instance, terrain data obtained using sensing technologies such as LiDAR, and GPS traces of vehicles consist of a collection of points on the earths surface, and points are the most basic geometric object. Many problems in other domains can also be viewed through a geometric

lens. For example, a SELECT query on a relational database can be viewed as asking what points lie inside a hyperrectangle in some high-dimensional space, an image can be mapped to a vector of pixel intensity values, and so on. The central theme of this thesis is to use geometric techniques to aid in the analysis of such data sets. The data sets available today are huge, and have sizes that were inconceivable not very long back. Many of these data sets also have a temporal aspect. For example, trajectories consist of locations along with timestamps; a video can be thought of as a sequence of images ordered temporally; observations of physical processes measured over time, etc. In some cases, so much data is generated at such a rapid rate that it cannot be stored in its entirety, and we can only make one pass over the *data stream*. Often the data available is noisy, inaccurate, incomplete, and contain outliers. This may happen for multiple reasons, e.g., sensor equipment malfunction, or lossy transmission of data. Such issues pose unique algorithmic challenges, and necessitate development of new techniques for data analysis.

This dissertation is mostly concerned with how to tackle some of the technical challenges associated with analyzing large spatio-temporal data. A major focus of this thesis is to develop algorithms with *provable performance guarantees*. Huge input sizes mean that it is no longer enough to design polynomial time algorithms, and we need (near-)linear time algorithms to make them scalable (when possible). The presence of noise in the data requires that the algorithms be robust and handle noise in a principled manner. Furthermore, many of the problems encountered are computationally hard to solve exactly. Since the input data itself is often noisy and an approximation of the actual data points, it is often sufficient to settle for algorithms that provide approximate answers but are much faster.

1.1 Broad Themes

We briefly outline the two broad approaches taken in this dissertation to get a handle on big noisy data sets.

Distributed algorithms. Many data sets are so huge that they cannot fit on one machine and are distributed across multiple machines, with a communication network linking the machines together. Such data sets are processed in parallel using big data platforms such as MapReduce [70], where a variety of issues arise ranging from communication latency to load balancing. Most algorithms for such platforms do not have any theoretical performance guarantees (see the survey [102] for a discussion on efficient algorithms for parallel query processing with provable guarantees on their performance). Designing provably efficient algorithms for such platforms demands renouncing traditional models of computation (such as the RAM model), and working with a model that abstracts away most of the implementation details of individual platforms yet captures the salient features of such platforms. We look at how to build data structures to answer range and nearest neighbor queries, and algorithms for analyzing massive terrain data, in a popular distributed model of computation.

Small-sized descriptors. For certain applications, maintaining the entire data set can be avoided, e.g., data sketching techniques such as Bloom filters and Count-Min have been pretty successful for summarizing sets and counting distinct items respectively [3]. In our case, we construct small-sized descriptors so that certain geometric and topological analyses can be performed efficiently using these descriptors, e.g., certain kinds of queries can be answered quicker using a contour tree (Section 3.2.2) than naively using the entire data set.

In many cases, we are interested in comparing two different data sets. One way

to do this is to compute correspondences between the two. For instance, geometric point set matching in two and three dimensions is a well-studied family of problems with applications to areas such as computer vision [124], pattern recognition [63, 98], and computational chemistry [83, 126]. Each correspondence is associated with a cost function that measures its *goodness*; the goal is to compute the *best* correspondence. Such correspondences also reveal shared substructures among the two data sets. Comparing the data sets directly can be difficult in some cases, hence these descriptors can be used as proxies for the data sets themselves, and we compare the descriptors directly. Further, these descriptors are robust to noise in the data, and using them for comparison is a way to compare data sets while handling noise in a principled manner.

The previous paragraph discussed comparing two objects. What if we want to find commonalities among multiple (more than two) objects? One way to do this is by clustering. The primary goal of any clustering algorithm is to group *similar* objects together. In many cases, huge data sets tend to cluster into smaller number of clusters. Uncovering the underlying clusters can reveal hidden structure in the data. Further, each cluster can sometimes be replaced by a *representative* that *captures* the entire cluster, thereby reducing the complexity of the data set. Such a representative can also lead to noise reduction, since it aggregates the noise present in each individual member of the cluster. Clustering can also help in finding outliers and anomaly detection.

Clustering only tells us which objects are the most similar. What if we want a more fine-grained approach to discovering common patterns in data? e.g., instead of looking at an object as a whole, we might want to look at sub-parts of it and compute similar sub-parts of other objects. This problem is different than clustering the objects in general, since two objects in different clusters may still share common portions. We specifically look at the problem of clustering *sub*-trajectories in input

trajectories.

1.2 Our Contribution

We describe our specific contributions briefly. This also serves as a guide to how the dissertation is organized.

1. We design algorithms for building and querying data structures to answer range and nearest-neighbor searching queries, in cases where the data is stored across multiple machines (Chapter 2). Such queries form the bedrock for several kinds of data processing tasks. We use the Massively Parallel Communication (MPC) model of computation (see Section 2.1) to analyze the performance of our algorithm, and prove that our algorithms are near-optimal (Theorems 10, 15 and 17). Our algorithms are simple and based on partitioning the input using a small-sized random sample. This chapter is based on joint work with Pankaj K. Agarwal, Kyle Fox and Kamesh Munagala [18].

2. We design MPC algorithms for two fundamental problems in terrain analysis – computing the Delaunay triangulation of a planar point set, and computing the contour tree of a triangulated terrain (Chapter 3). A triangulation of a planar point set is said to be Delaunay if the circumcircle of any of its triangles does not contain any input point in its interior. The contour tree is a data structure that succinctly encodes the contours of a real-valued function. We prove that our algorithms are optimal for the Delaunay triangulation (Theorem 28), and are optimal for the contour tree under certain reasonable assumptions on the input (Theorem 33). We use a divide-and-conquer approach that takes advantage of the underlying geometry to split the problem into constituent sub-problems, while reducing the *interaction* between *neighboring* sub-problems. This chapter is based on joint work with Pankaj K. Agarwal, Kyle Fox and Kamesh

Munagala [125].

3. We explore the Gromov-Hausdorff (GH) distance between metric spaces (Chapter 4). The GH distance measures the smallest additive *distortion* that can be achieved while embedding one metric space into another. The problem is challenging because it is closely related to the *graph isomorphism problem*, which is known to be neither polynomial-time solvable, nor NP-complete. We prove hardness results (Theorem 36), and give approximation algorithms for computing the GH distance in the special case of tree metrics (Corollary 52). We reduce the problem to comparing merge trees of real-valued functions, another interesting problem in its own right. This chapter is based on joint work with Pankaj K. Agarwal, Kyle Fox, Anastasios Sidiropoulos and Yusu Wang [21].

4. We look at the problem of extracting common movement patterns from trajectory data (Chaper 5). We cast this as a *subtrajectory clustering* problem. We propose a new model for clustering subtrajectories, and provide hardness results (Theorems 53 and 54) and approximation algorithm for the same (Theorem 56). We further show that our algorithm can be made much faster under some realistic input assumptions (Theorem 67). Our algorithms use a reduction to the set cover problem; however the size of the set cover instance is exponential and hence cannot be constructed explicitly. The main challenge is then to implement the greedy set cover algorithm efficiently without ever constructing the set cover instance explicitly. This chapter is based on joint work with Pankaj K. Agarwal, Kyle Fox, Kamesh Munagala, Jiangwei Pan and Erin Taylor [19].

1.3 Prior Work

We discuss previous work, and refer the reader to the respective later chapters for more details.

Big data platforms. The term "big data" has become ubiquitous to describe data that cannot be stored or processed on a single machine. The MapReduce platform [70], its open source implementation Hadoop [145], and related platforms (such as Pregel [114], Spark [148], and Google Cloud Dataflow [26]) have emerged as dominant computing platforms for big-data processing. At a high level, these systems focus on computation local to the data stored on individual machines and have become popular due to their ability to abstract away the distributed nature of data storage and processing. The data itself is stored on disk in a distributed file system, for example, the Hadoop Distributed File System (HDFS) of the Hadoop platform [136]. A distributed file system can be treated as a cluster of machines, each with its own disk space where the data is stored. Spark [148] builds on MapReduce and attempts to keep data in memory to speed up machine learning applications that require multiple passes over the same data. Many of these platforms also incorporate streaming and real-time primitives. A high level programming language such as Pig [4] is used to write a program that is pushed to the machines where the data is stored and processed.

Theoretical models for big data and parallel computation. Theoretical models of computation capture the salient features of real-world systems while abstracting away implementation details, thereby simplifying the process of designing and analyzing algorithms.

A common concern when designing data structures for massive amounts of data is the cost of reading and writing from disk, called an I/O operation. Aggarwal

and Vitter [23] described the two-level I/O-memory model in an attempt to understand those costs. An alternative to the two-level I/O model described above is the cache-oblivious model introduced by Frigo *et al.* [84]. In essence, algorithms for the traditional RAM model are evaluated in terms of performance using the two-level I/O model with *unknown* internal memory size and unknown block size. The analysis assumes transfers between internal and external memory are done using an optimal caching strategy.

While the I/O models above are designed with the storage of massive data in mind, they are still inherently *sequential*. In contrast, the Parallel Random Access Machine (PRAM) model ignores I/O complexities and instead models parallel processors sharing a common memory pool. The PRAM model neglects such issues as synchronization and communication, and the cost of an algorithm is estimated using the overall running time as well as the sum of the running times of all processors.

The Bulk Synchronous Parallel (BSP) model [141] of Valiant models parallel processing, communication, and synchronization. A specialization of this model to multiprocessors is termed coarse grained parallelism (CGP) [71]. This model is similar to MPC; there are $p \leqslant n^{\varepsilon}$ processors (where n is the input size and $\varepsilon \leqslant 0.5$), each of which is allowed $O(n/p)$ communication between rounds, and arbitrary sequential computation. The goal is to simultaneously optimize the number of rounds, as well as the sequential computation done per processor.

More recently, many authors have focused on variations of the MapReduce model [105, 100, 87, 91, 41, 28]. All of these models involve multiple machines with the caveat that no machine can hold all of the input at once. Computation proceeds in rounds, with each machine having access to only its own data in any round. The machines can communicate with each other between rounds. Briefly, let ε be such that $0 < \varepsilon < 1$, and let n denote the input size. Karloff *et al.* [100] restrict both the number of machines and the memory on each machine to be $O(n^{1-\varepsilon})$, and they only consider

the number of rounds taken by an algorithm as its measure of performance. Further they restrict the total amount of communicated between rounds to $O(n^{2-2\varepsilon})$. This model is used by Lattanzi *et al.* [105] to design certain graph theoretic algorithms; however they also take into account the total time taken by all the machines in each round to evaluate their performance. The models of [87, 91, 41] are more or less similar – they assume that if there are p machines then each machine has space $s = O(n/p)$; they are also interested only in the number of rounds of computation. Andoni *et al.* [28] further restrict the model of [41] so that $s \geqslant \sqrt{n}$, the justification being that the space on each machine is much larger than the number of machines in practice. There is extensive work in database systems on developing algorithms under MapReduce and its variants, e.g., algorithms for large graph processing, join operations, query processing etc. See e.g. [132, 99, 107].

I/O-efficient and parallel geometric algorithms. Extensive work has been done on developing range-searching data structures in the I/O model [30] and the cache-oblivious model [31]. In addition, Agarwal *et al.* [13] describe I/O-optimal algorithms for constructing range and nearest-neighbor searching data structures. Many efficient geometric algorithms in the PRAM model have been described; see Goodrich [90] for a survey. Dehne *et al.* [71] present optimal algorithms for several geometric problems in the CGP model.

As far as practical implementations are concerned, there are several MapReduce implementations for analyzing and querying spatial and geometric data, see [77, 27, 79, 78, 12] and references there in. For example, SpatialHadoop [79] is a full-fledged MapReduce framework that adapts traditional spatial index structures like R-tree and R+-tree to form a two-level spatial index. It is also equipped with other operations, including range query, k-nearest neighbors, and spatial join.

Goodrich *et al.* [91] describe a connection between MapReduce and the BSP

model mentioned above, which leads to efficient implementation of some geometric algorithms. Andoni *et al.* [28] develop MapReduce algorithms for approximating the minimum spanning tree and the earth-mover distance in a bounded doubling dimension metric space. There is some work on similarity search in high dimensions in the distributed setting [36]. However, their focus is on reducing the amount of communication per round and is based on distributed locality-sensitive hashing.

Comparing metric spaces and topological descriptors. Most work on associating points between two metric spaces involves *embedding* a given high dimensional metric space into an infinite host space of lower dimensional metric spaces. However, there is some work on finding a bijection between points in two given finite metric spaces that minimizes typically multiplicative distortion of distances between points and their images, with some limited results on additive distortion.

Kenyon *et al.* [101] give an optimal algorithm for minimizing the multiplicative distortion of a bijection between two equal-sized finite metric spaces, and a parameterized polynomial time algorithm that finds the optimal bijection between an arbitrary unweighted graph metric and a bounded-degree tree metric.

Papadimitriou and Safra [129] show that it is NP-hard to approximate the multiplicative distortion of any bijection between two finite 3-dimensional point sets to within any additive constant or to a factor better than 3.

Hall and Papadimitriou [94] discuss the *additive distortion problem* – given two equal-sized point sets $S, T \subset \mathbb{R}^d$, find the smallest Δ such that there exists a bijection $f : S \to T$ such that $d(x, y) - \Delta \leqslant d(f(x), f(y)) \leqslant d(x, y) + \Delta$. They show that it is NP-hard to approximate by a factor better than 3 in $\mathbb{R}^3$, and also give a 2-approximation for $\mathbb{R}^1$ and a 5-approximation for the more general problem of embedding an arbitrary metric space onto $\mathbb{R}^1$.

The interleaving distance between merge trees [121] was proposed as a measure

to compare functions over topological domains that is stable to small perturbations in a function. The interleaving distance can be defined for more general settings, and is a popular distance measure in topological data analysis; see [43] for more recent results on its computational complexity. Distances for the more general Reeb graphs are given in [38, 69]. These concepts are related to the Gromov-Hausdorff distance between metric spaces [92].

The field of shape analysis involves data acquisition and shape modeling. Hence defining and computing meaningful notions of similarity between such digital models is very important. Such data is usually in the form of a point cloud and are endowed with a notion of distance between its points. Thus they can be viewed as metric spaces. In many cases, the objects are invariant under various deformations, and as such it is important to come up with an *intrinsic* distance measure between metric spaces that is independent of the ambient metric space in which the points reside; see [59, 48, 118, 119].

Discovering shared structure in trajectories and other sequence data. There has been a recent line of work on subtrajectory clustering [62, 93, 106, 137, 150, 50, 128, 134]. Apart from these, there is a fair bit of work on extracting common movement patterns. For example, given a set of trajectories and a query time-interval, Pelekis *et al.* [130, 131] cluster the subtrajectories having points lying in the query interval. This is equivalent to clustering trajectories, obtained by restricting each trajectory in the query interval. The algorithms in [82, 144, 149, 138] search for pre-defined patterns, such as groups or crowds, in trajectory data.

The work on multiple sequence alignment (MSA) in computational biology [127], on functional clustering in statistics [133], and on topic modeling/dictionary learning [44, 76] in machine learning and signal processing also focus on identifying shared structures.

Finally, there is work on computing a mean or a median trajectory as a representative of a given set of "similar" trajectories [51], computing basis trajectories to recover structure from motion [25], and segmenting individual trajectories using certain geometric criteria [53]. The work on trajectory clustering [85, 97, 146] focuses on partitioning a set of trajectories into clusters of similar trajectories, and possibly computing a representative trajectory for each cluster using the aforementioned work. This line of work, however, does not discover shared structures among otherwise dissimilar trajectories. There is some work on partial matching between a pair of trajectories, i.e., finding most similar subtrajectories between two trajectories [52].

2

Querying Massive Data

2.1 Introduction

One of the important problems in databases, GIS, and computational geometry is to answer various queries on a set of points in a geometric space. One could conceivably scan the entire data to answer each query, but since several queries are answered on the same data, it is desirable to preprocess data into a data structure so that a query can be answered quickly. A popular approach to cope with big data in the context of query processing is to work with a small summary of data such as random samples, coresets, sketches, *etc.* These methods are successful for answering aggregation queries, but they do not work well when queries involve analyzing local structure of data such as nearest-neighbor queries or range-reporting queries (especially for small ranges). In such instances, one has to work with the entire data. This difficulty raises the problem of constructing data structures on big data platforms. In this chapter, we study data structures for range-reporting and nearest-neighbor queries – two very popular queries on geometric data – on big data platforms. In particular, we develop efficient algorithms for constructing some of the classical data structures

such as *kd*-trees, BBD-trees, and range trees.

Our model. To avoid dependence on specifics of a particular platform, we present our algorithms in a simple but popular abstract model called the (basic) *massively parallel communication* (MPC) model, originally proposed by Beame *et al.* [41] (see also Andoni *et al.* [28]).

Let n be the size of an input instance, and let I be a set of m machines. For simplicity, we assume I to be $\{0, \ldots, m-1\}$. Set $s = n/m$, and for simplicity, assume s is an integer.[1] Each machine has $O(s)$ memory (or space). As is standard, we assume $s \geqslant n^{\alpha}$ for some positive constant $\alpha < 1$. Computation proceeds in rounds. In each round, each machine reads its input, does some computation, and emits some output, where each output item is marked with the ID of the machine for which it is input in the next round. The size of the input to and output from any machine is bounded by $O(s)$ in each round. Communication across machines occurs only between rounds. See Figure 2.1.

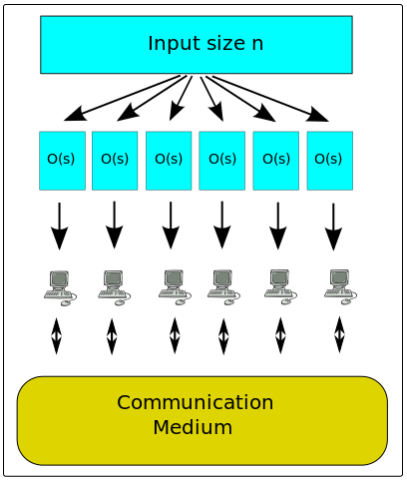


FIGURE 2.1: The MPC model.

We explicitly allow data to persist on machines between rounds of computation

[1] For simplicity, we will ignore floor and ceiling operators throughout this chapter and assume appropriate choice of parameters so that the ratios are integers when needed.

and after all computation has been performed, as long as the total amount of data stored on each machine never exceeds $O(s)$. By considering data storage as we do, we are able to build and store data structures for massive geometric data. The explicit persistence of data between rounds and our hard requirements on the space of each machine are the main differences between our model and the MPC model as described by Beame *et al.* [41]. Andoni *et al.* [28] also limit the space on each machine, but they do not explicitly consider persistent storage. In fact, some work in similar models (*e.g.* Goodrich *et al.* [91]) require machines to communicate their own data to themselves in order for data to persist between rounds of computation.

Queries to our data structures behave as any other MPC computation. They simply take advantage of the distributed data structures already stored in the machines to reduce query time. When preprocessing a set of points P to build our data structures, we assume the points of P are distributed arbitrarily throughout the machines. Individual queries are sent to an arbitrary machine.

The efficiency of an algorithm is measured using three metrics: the number of rounds of computation R, the running time T, and the total work W. The first is obvious; we describe the latter two below. For machine $\beta \in I$, let $t_{\beta r}$ denote the computation time spent by this machine in round r. The *running time* is defined as

$$T = \sum_{r=1}^{R} \max_{\beta \in I} t_{\beta r}.$$

Note that this definition does not count the time it takes to communicate between the machines, which is accounted for separately by the quantity R.

The *total work* is defined as

$$W = \sum_{r=1}^{R} \sum_{\beta \in I} t_{\beta r}.$$

In order to make guarantees about our total work, we assume a machine performs

zero work in a single round of computation if it does not receive any input or communication at the beginning of that round. This assumption can be interpreted as each machine "sleeping" through a round of computation unless it is given a reason to wake up, and this makes it possible for W to be significantly smaller than $m \cdot T$.

While most prior work on the MPC model focuses on minimizing the number of computation rounds R of an algorithm, e.g., [41, 100, 80], we must consider all three metrics in order to say anything interesting about geometric data structures. Consider processing a set of points $P \subseteq \mathbb{R}^2$ so that one may report the points lying within an axis-aligned query rectangle. If number of rounds were our only concern, we would build a trivial data structure in one round of computation by doing nothing. A query for rectangle $\square_q$ would then be performed in one round of computation as well by having each machine individually check which of its points lie inside $\square_q$. However, queries will take $O(s)$ time and require $O(n)$ work. We could slightly improve our data structure by building kd-trees for each machine's set of points using one round, $O(s \log s)$ time, and $O(n \log n)$ total work (see Section 2.3). The runtime of a query would improve to $O(\sqrt{s})$ in the worst case, but queries would still require $\Omega(m)$ work in the best case and $O(\sqrt{s}m) = O(n/\sqrt{s})$ work in the worst case. Unless s is nearly as large as n, these worst-case workloads are significantly higher than what is possible using classical sequential data structures. Therefore, we will focus on minimizing all three performance metrics.

2.1.1 Our results

Let P be a set of n points in $\mathbb{R}^d$, where d is a constant. We present efficient algorithms for building and querying kd-trees, range trees, and BBD-trees on P. The first two are used for range queries and the last one is used for approximate nearest-neighbor (NN) queries. They are formally defined in later sections; see [68, 34] for details. A kd-tree uses linear space and $O(n^{1-1/d} + k)$ query time to answer an orthogonal

range query (*i.e.*, reporting all k points of P lying in a query axis-parallel rectangle). It can be computed in $O(n \log n)$ time. In contrast, a range tree uses $O(n \log^{d-1} n)$ space and $O(\log^{d-1} n + k)$ query time; range trees with slightly smaller space are also known [17]. A BBD-tree can answer an ε-approximate NN query in $O(c_{d,\varepsilon} \log n)$ time using $O(n)$ space where $c_{d,\varepsilon}$ is a constant dependent on d and ε.

For all of the data structures we consider, we will show a bound of $R = \text{polylog}_s\, n = O(1)$ on the number of rounds of computation required both to build the data structure and to perform queries. Our running times T for building our data structures and performing queries will be comparable to the best sequential algorithms when performed on point sets of size s and our total work W will be comparable with the best sequential algorithms for point sets of size n. More precisely, we obtain the following results; see Table 2.1.1 for a summary.

- A *kd*-tree on P can be built in $O(1)$ rounds, $O(s \log s)$ time, and $O(n \log n)$ work with probability at least $1 - \frac{1}{n^{\Omega(1)}}$.[2] Queries can be answered using $O(1)$ rounds and $O(n^{1-1/d} + k)$ work where k is the output size; the running time is $O(s^{1-1/d} + k')$ where k' is the maximum number of points reported by a machine.

- A BBD-tree on P can be built in $O(1)$ rounds, $O(s \log s)$ time, and $O(n \log n)$ work with probability at least $1 - \frac{1}{n^{\Omega(1)}}$. Queries can be answered in $O(1)$ rounds, $O\left(\frac{1}{\varepsilon^d} \log\left(\frac{1}{\varepsilon}\right) \log n\right)$ time, and $\frac{1}{\varepsilon^{O(d)}} \log n$ work.

- A range tree on P can be built in $O(1)$ rounds, $O(s \log^d s)$ time, and $O(n \log^d n)$ work.[3] Queries can be answered in $O(\log^d n + k')$ time and $O(\log^d n + k)$ work in

[2] The *kd*-tree we build is a slight variant of the standard *kd*-tree in that unlike the latter, our tree is not exactly balanced in partitioning the points. Nevertheless, we show that the space, query, and preprocessing requirements are asymptotically identical.

[3] In order to have sufficient space just to store the range tree, we slightly amend our model so the memory and per-round input and output size of each machine is $O(s \log^{d-1} n)$.

Performance metric		kd-tree	BBD-tree	Range tree
Pre-processing	Rounds	$O(1)$	$O(1)$	$O(1)$
	Time	$O(s \log s)$	$O(s \log s)$	$O(s \log^d s)$
	Work	$O(n \log n)$	$O(n \log n)$	$O(n \log^d n)$
Query	Rounds	$O(1)$	$O(1)$	$O(1)$
	Time	$O(s^{1-1/d}+k')$	$O(\log n)$	$O(\log n + k')$
	Work	$O(n^{1-1/d}+k)$	$O(\log n)$	$O(\log^d n+k)$

Table 2.1: Our results for building and querying distributed data structures.

$O(1)$ rounds of computation, where k' and k are the maximum number of points reported by a machine and the total number of points reported respectively.

The algorithm for kd-trees can be extended to many variants of kd-trees such as hB-trees [113], hB^{π}-trees [81], and box trees [16]. It can also be extended to construct a partition tree [58] that is used for answering simplex range searching queries, *i.e.*, reporting all points that lie inside a query simplex.

The algorithms for building kd-trees and BBD-trees use randomization. However, they can be derandomized while keeping the number of computation rounds constant by increasing the running time – we obtain a tradeoff between the number of rounds and the running time. Specifically, for a parameter $\beta \leqslant 1/5$, the data structures can be constructed deterministically in $O(1/\beta)$ rounds, $s^{1+O(\beta)}$ time, and $n^{1+O(\beta)}$ work.

We remark that there is extensive work on I/O-efficient data structures in databases. For example, the kdB-tree is an I/O-efficient version of a kd-tree [30]. Similarly I/O-efficient range trees have been proposed [30]. Our algorithms can be modified easily to construct I/O-efficient versions of the data structures.

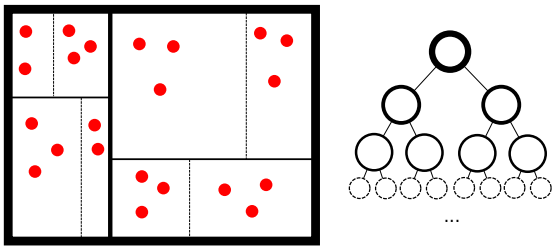


FIGURE 2.2: A *kd*-tree partitions space into a hierarchy of boxes. Each box is represented by a node in a tree, and nesting boxes are represented by ancestor-descendant relationships within the tree.

2.1.2 Our techniques.

Our algorithms for constructing data structures depend upon the concept of ε-approximations defined formally in Section 2.2.1. Intuitively, an ε-approximation is a small subset of points $S \subset P$ such that any low complexity region $\mathcal{R}$ of $\mathbb{R}^d$ contains about the same fraction of points of S as that of P. One property common to most of the data structures we build is that they induce a hierarchical decomposition of $\mathbb{R}^d$ represented by a balanced tree. See Figure 2.2. In a constant number of rounds of computation, we sample an ε-approximation S of P. We choose S to be small enough to fit on a single machine, so we use a sequential algorithm to build the first few layers of the tree data structure using only points in S. The regions of $\mathbb{R}^d$ associated with the leaves of the partial tree partition S into approximately equal sized subsets. Because S is an ε-approximation of P, these same regions partition P into almost equal sized subsets as well. We recursively build the lower levels of the data structure on each piece of the partition.

Our algorithms are simple, and our analysis shows that a simple sampling based technique gives essentially the best possible running time in the MPC model, without resorting to additional features that some modern data processing techniques may enable.

Sampling and the similar concept of filtering have been used before for the design

of efficient algorithms for models based on MapReduce [80, 79, 105, 104]. However, these algorithms complete their work on a single machine after acquiring a small set of samples. In contrast, the partial trees our algorithms compute using their sample sets are not an entire solution; the algorithms must continue processing the remaining points in the input set using the partial trees to aid in the remaining computation.

2.1.3 Related work

The I/O-optimal algorithms for constructing *kd*-trees and BBD-trees by Agarwal *et al.* [13] are not easily parallelizable. Although Dehne *et al.* [71] solve many geometric problems optimally in a model of computation similar to MPC, they use a simple deterministic partitioning of the point sets, and do not focus on the more challenging problem of constructing data structures. The geometric approximation algorithms of Andoni *et al.* [28] under MapReduce use a partitioning scheme based on randomly shifted quadtrees, hence largely independent of the input point set. On the other hand, we use a small sample of the input points in a clever way to partition the points. The work on similarity search in high dimensions in the distributed setting [36] is based on distributed locality-sensitive hashing, whereas our approach is quite different and is tailored for low dimensions.

The practical MapReduce implementations for analyzing and querying spatial and geometric data [77, 27, 79, 78, 12] do not have any *provable bounds* on their performance, which is the main focus of this chapter.

The rest of the chapter is organized as follows. We describe some primitive operations useful for building geometric data structures in Section 2.2. We discuss *kd*-trees and an extension of our techniques to partition trees [58] in Section 2.3. We discuss BBD-trees and range trees in Sections 2.4 and 2.5, respectively. Finally, we conclude in Section 2.6 by mentioning some directions for future research.

2.2 MPC Primitives

Before describing our MPC primitives, we introduce the notion of an ε-approximation, which will be constructed by one of the primitives and which will be used by many of our algorithms.

2.2.1 Range spaces and ε-approximations

A *range space* Σ is a pair $(X, \mathcal{R})$, where X is a ground set and $\mathcal{R}$ is a family of subsets (*ranges*) of X. For example, X is a set of points in $\mathbb{R}^2$ and

$$\mathcal{R} = \{X \cap \square \mid \square \text{ is a rectangle in } \mathbb{R}^2\}.$$

A subset $X' \subseteq X$ is *shattered* by Σ if $\{X' \cap R \mid R \in \mathcal{R}\} = 2^{X'}$. The *VC dimension* of Σ is the size of the largest subset of X shattered by Σ. If there are arbitrarily large shattered subsets, the VC dimension of Σ is set to ∞.

Given range spaces $\Sigma_1 = (X, \mathcal{R}_1)$ and $\Sigma_2 = (X, \mathcal{R}_2)$ with VC dimensions δ_1 and δ_2 respectively, the union of Σ_1 and Σ_2 is defined as

$$(X, \mathcal{R}) = (X, \{R_1 \cup R_2 \mid R_1 \in \mathcal{R}_1, R_2 \in \mathcal{R}_2\}),$$

and has VC dimension $O(\delta_1 + \delta_2)$. The complement of Σ is defined as $(X, \overline{\mathcal{R}}) = (X, \{X \backslash R \mid R \in \mathcal{R}\})$ and has the same VC dimension as Σ. See Chazelle [60] for details.

Given a range space $\Sigma = (X, \mathcal{R})$ and $0 \leqslant \varepsilon \leqslant 1$, a subset $X' \subseteq X$ is called an *ε-approximation* of Σ if for any range $R \in \mathcal{R}$, we have

$$\left| \frac{|X' \cap R|}{|X'|} - \frac{|R|}{|X|} \right| \leqslant \varepsilon.$$

The following bound on an ε-approximation was proved by Li *et al.* [111].

Theorem 1 ([111]). *There is a positive constant c such that if $\Sigma = (X, \mathcal{R})$ is a range space with VC dimension δ, then a random subset of X of size $\frac{c}{\varepsilon^2}\left(\delta + \log\frac{1}{\psi}\right)$ is an ε-approximation of X with probability at least $1 - \psi$.*

For range spaces with constant VC dimension, a random subset of size $O\left(\frac{1}{\varepsilon^2}\log n\right)$ is an ε-approximation with probability at least $1 - 1/n^{\Omega(1)}$, where n is the size of the ground set.[4] The ranges that we consider in this chapter are induced by axis-aligned boxes, simplices, and rectilinear regions defined by constant number of rectangles. Since these regions can be formed by taking a Boolean combination of a constant number of halfspaces, the range spaces induced by these regions have constant VC dimension.

2.2.2 Geometric primitives and techniques

We define a few primitive operations that we use to build our distributed data structures.

`PrefixSum`$(P, I, \text{rank}, \text{value})$**:** Given a set of points P stored on a contiguous subset of machines $I = \{i_0, i_0 + 1, \dots\}$, a rank function $\text{rank} : P \to \mathbb{Z}^+$, and a value function $\text{value} : P \to \mathbb{R}$, compute the prefix sum of values for each point $p \in P$ where $\text{rank}(p)$ is the rank of point p in some sorted order.

`Broadcast`$(\mathcal{S}, I, \beta)$**:** Given a set of words $\mathcal{S}$, a contiguous set of machines $I = \{i_0, i_0 + 1, \dots\}$, and a machine β storing $\mathcal{S}$, copy $\mathcal{S}$ to all machines in I. We require $\mathcal{S}$ has size $O(s^{1/2})$.

`Partition`(P, I, Π, β)**:** Given a spatial subdivision Π of $\mathbb{R}^d$ into simple regions (e.g., rectangles, simplices), reorganize the points of P so that all points lying in a cell π of Π lie on a distinct contiguous subset of machines $I_\pi \subseteq I$. See Figure 2.3.

[4] Better bounds for the size of ε-approximations exist for range spaces with finite VC dimension [60], but the bounds given here suffice for our purposes.

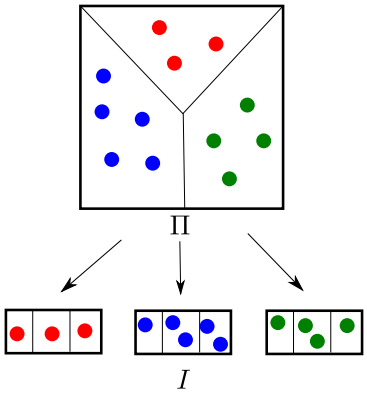


FIGURE 2.3: `Partition`(Π, I, β) reorganizes the points stored on I as dictated by the partition Π.

We assume I itself is also a contiguous subset of machines $\{i_0, i_0 + 1, \dots\}$. We require that $|\Pi| = O(s^{1/2})$, $|\Pi| \leqslant |I|$, and each cell π of Π contains $O(|P|/|\Pi|)$ points.

`Sample`(P, I, r, β)**:** Given a set of points P stored on a contiguous subset of machines $I = \{i_0, i_0 + 1, \dots\}$, compute a $(1/r)$-approximation $S \subseteq P$ of size $O(r^2 \log n)$ and send it to machine β. We require $|S| = O(s^{1/2})$.[5]

Before describing how to implement these primitives efficiently, we describe a procedure that will be used by these primitives.

To facilitate transfer of information between a given contiguous subset of machines $I = \{i_0, i_0 + 1, \dots\}$, we construct a tree T, a *complete* $s^{1/2}$*-ary tree* with $|I|$ leaves. Each node v of T is associated with a single machine $\beta(v) \in I$ such that each machine is used at most once per level of T. We let r_T denote the root of T and set $\beta(r_T)$ appropriately depending on the primitive. Let λ_T denote the depth of T. We have $\lambda_T = O(\log_{s^{1/2}} |I|) = O(1)$. Goodrich *et al.* [91] use a similar tree to compute prefix sums. A number of simple operations can be completed in a constant number

[5] Strictly speaking, the algorithm for computing a $(1/r)$-approximation depends on the underlying range space. In our applications, the range space associated with P will be clear from the context and will have constant VC dimension.

of rounds of computation guided by T, by passing information between machines belonging to parent and child nodes. For example, it is straightforward to compute the max and sum of several numbers stored across the machines. While sending information from parent to children, the parent node's machine sends $O(s^{1/2})$ data to each child node's machine. If information is sent from children to parent, each child's machine sends $O(s^{1/2})$-size data to the parent's machine. Since each node has $s^{1/2}$ children, the input and output size for each machine is bounded by $O(s)$ in one round.

We now describe an implementation of above primitives.

`PrefixSum`$(P, I, \text{rank}, \text{value})$**:** It can be implemented in $O(1)$ rounds with linear time and work using an algorithm of Goodrich *et al.* [91]. Without giving details here, we state the bounds :

Lemma 2. `PrefixSum`$(P, I, \text{rank}, \text{value})$ *can be implemented by a deterministic algorithm in* $O(1)$ *rounds,* $O(s)$ *time, and* $O(n)$ *total work.*

`Broadcast`$(\mathcal{S}, I, \beta)$**:** We use the tree T over the machines in I. Our broadcast procedure takes exactly $\lambda_T + 1 = O(1)$ rounds of computation. In round i, each machine $\beta(v)$ for a node v at level $i-1$ receives a copy of $\mathcal{S}$. If it has not done so already, $\beta(v)$ stores a copy of $\mathcal{S}$. When the round of computation ends, $\beta(v)$ sends a copy of $\mathcal{S}$ to each machine $\beta(v')$ where v' is a child of v in T.

Lemma 3. `Broadcast`$(\mathcal{S}, I, \beta)$ *can be implemented by a deterministic algorithm using* $O(1)$ *rounds of computation,* $O(s)$ *time, and* $O(n)$ *total work.*

`Partition`(P, I, Π, β)**:** Order the cells of Π in an arbitrary way, and let $\{\pi_0, \pi_1, \dots\}$ be the cells of Π. We let $I_{\pi_k} = \{i_0 + k(|I|/|\Pi|), \dots, i_0 + (k+1)(|I|/|\Pi|) - 1\}$. We begin by running `Broadcast`(Π, I, β) to store Π on all the machines. In order to distribute the points of each partition cell π evenly across I_π, we use the `PrefixSum` primitive to

count for each machine-cell pair (i, k) the points that are either in cells $\{\pi_0, \ldots, \pi_{k-1}\}$ or in cell π_k and stored on machines $\{i_0, \ldots, i-1\}$. Using these counts, we can then quickly finish the partitioning procedure.

Each machine $i \in I$ creates a set of $|\Pi|$ *counting points* C_i to be given to the `PrefixSum` primitive. The point $q_{i,k} \in C_i$ is associated with the cell π_k. Let $\text{rank}(q_{i,k}) = k|I| + i - i_0 + 1$, and let $\text{value}(q_{i,k})$ be equal to the number of points on machine i in cell π_k. Let $C = \bigcup_{i \in I} C_i$. We run $\texttt{PrefixSum}(C, I, \text{rank}, \text{value})$. The prefix sum of counting point $q_{i,k}$ is the number of points in cell π_k stored on machines $\{i_0, \ldots, i-1\}$ plus the number of points in cells $\{\pi_0, \ldots, \pi_{k-1}\}$. Now, let $\{p_0, p_1, \ldots\} \subseteq P$ be a set of points in arbitrary order belonging to some machine i and partition cell π_k. Let $q_{|I|+1,k} = q_{|I|,k} + \text{value}(|I|, k)$. Each value $q_{i+1,k}$ can be sent to machine i in one round of communication. Machine i then sends point p_j to machine $i_0 + k(|I|/|\Pi|) + \lfloor (q_{i+1,k} - q_{i,k} + j)/(|P|/|\Pi|) \rfloor$.

Lemma 4. $\texttt{Partition}(P, \Pi, I, \beta)$ *can be implemented by a deterministic algorithm using* $O(1)$ *rounds of computation,* $O(s \log s)$ *time, and* $O(n \log n)$ *total work.*

$\texttt{Sample}(P, I, r, \beta)$**:** We first describe a simple Las Vegas algorithm for computing the sample $S \subseteq P$. Each machine i selects a subset of its points S_i by selecting points independently, each with probability $p = c(r^2/|P|) \ln n$ for some constant c that depends on the VC dimension of the underlying range space. We then use the tree T to compute the total *number* of selected points in $\lambda_T + 1 = O(1)$ rounds of computation by summing the values $|S_i|$ across each machine. If this number is more than $2cr^2 \ln n$ or less than $(c/2)r^2 \ln n$, an incorrect number of points have been selected and the procedure restarts. Otherwise, all the selected points from each machine are sent to β to form the set S. A standard Chernoff bound [120] guarantees the probability of restarting even once is at most $\exp(-(c/8)r^2 \ln n) \leqslant 1/n^2$ for large enough c. Each sample of size $|S|$ is chosen with equal probability, so S is an $(1/r)$-

approximation with probability at least $1 - 1/n^2$ as well by Theorem 1.

We can also choose S deterministically as follows. We use the tree T and let $\beta(r_T) = \beta$. Our sampling procedure still takes $\lambda_T + 1 = O(1)$ rounds of computation. In round i, each machine $\beta(v)$ for a node v at depth $\lambda_T - i + 1$ receives a set S_v of $O(s)$ points from its children that may be used in the $(1/r)$-approximation. If v is a leaf of T, then $\beta(v)$ simply uses the members of P initially found at $\beta(v)$. Machine $\beta(v)$ computes a (c/r)-approximation $S'_v \subseteq S_v$, for some sufficiently small constant c, of size $O(r^2 \log n) = O(s^{1/2})$ using the deterministic algorithm of Matoušek [116] in $O(s(r^2 \log r)^\delta)$ time. If $v = r_T$, then S'_v is the final $(1/r)$-approximation desired by the algorithm. When the round of computation ends, $\beta(v)$ sends set S'_v to $\beta(p(v))$, the machine corresponding to the parent of v.

Lemma 5. *The above procedure computes a $(1/r)$-approximation of P.*

Proof. Let P_v be the subset of points contained in the subtree of T rooted at v, and let λ_v be the depth of v. Fix a node v, and assume inductively that $S'_{v'}$ is a $(\gamma^{\lambda_{v'} - \lambda_T} c/r)$-approximation of $P_{v'}$ for each descendant v' of v for some constant γ. The children of v can be divided into three groups: the children where every descendant leaf has depth λ_T, children where every descendant leaf has depth $\lambda_T - 1$, and the solitary child with descendant leaves of both depths. Let V_1, V_2, and V_3 be these respective groups of child nodes. For any pair of children v_1, v_2 in a single group, there exist constants c' and c'' such that $c'|P_{v_2}| \leqslant |P_{v_1}| \leqslant c''|P_{v_2}|$ and $c'|S'_{v_2}| \leqslant |S'_{v_1}| \leqslant c''|S'_{v_2}|$. Let v' be an arbitrary node of V_i, and let $x_i = |S'_{v'}|$ and $y_i = |P_{v'}|$.

Let R be an arbitrary range in our range space. We have

$$\begin{aligned}
&\left|\frac{|S_v \cap R|}{|S_v|} - \frac{|P_v \cap R|}{|P_v|}\right| \\
\leqslant &\sum_{i=1}^{3} \left|\frac{|(\bigcup_{v' \in V_i} S'_{v'}) \cap R|}{|\bigcup_{v' \in V_i} S'_{v'}|} - \frac{|(\bigcup_{v' \in V_i} P_{v'}) \cap R|}{|\bigcup_{v' \in V_i} P_{v'}|}\right| \\
\leqslant &\sum_{i=1}^{3} \left|\frac{|(\bigcup_{v' \in V_i} S'_{v'}) \cap R|}{c'|V_i|x_i} - \frac{|(\bigcup_{v' \in V_i} P_{v'}) \cap R|}{c''|V_i|y_i}\right| \\
\leqslant &\sum_{i=1}^{3} \frac{1}{|V_i|} \sum_{v' \in V_i} c''' \left(\left|\frac{|S'_{v'} \cap R|}{|S'_{v'}|} - \frac{|P_{v'} \cap R|}{|P_{v'}|}\right|\right) \\
\leqslant &\gamma^{\lambda_v - \lambda_T} c/2r
\end{aligned}$$

where c''' is a constant.

Therefore, S_v is a $(\gamma^{\lambda_v - \lambda_T} c/2r)$-approximation of P_v. Set S'_v is a $(\gamma^{\lambda_v - \lambda_T} c/r)$-approximation [116, Observation 4.3]. T has depth $O(1)$, so S'_{r_T} is a $(1/r)$-approximation of $P_{r_T} = P$ when c is sufficiently small.[6] □

Lemma 6. *(i)* $\texttt{Sample}(P, I, r, \beta)$ *can be implemented by a Las Vegas algorithm using* $O(1)$ *rounds of computation,* $O(s)$ *time, and* $O(|P|)$ *total work, with probability at least* $1 - 1/n^{\Omega(1)}$.

(ii) The procedure can be implemented by a deterministic algorithm using $O(1)$ *rounds of computation,* $O(s(r^2 \log r)^\delta)$ *time, and* $O(|P|(r^2 \log r)^\delta)$ *total work.*

2.3 Constructing *kd*-tree

Given a set of n points $P \in \mathbb{R}^d$, a *kd*-tree on P can answer an orthogonal range-reporting query in $O(n^{1-1/d} + k)$ time using $O(n)$ space, where k is the number of points lying inside the query rectangle. Each node v of the tree is associated with a

[6] Our procedure and its proof require a fairly small constant c. One can increase the constant by guaranteeing each set $P_{v'}$ has equal size and each set $S_{v'}$ has equal size by choosing T carefully and adding $O(n)$ additional points. While possible, doing so would be very tedious in our model.

d-dimensional rectangle $\square_v$, called the cell of v, with the root cell being large enough to contain the entire set P. Let $P_v = P \cap \square_v$. If $|P_v| \leqslant 1$, v is a leaf and P_v is stored at v. If $|P_v| > 1$, then $\square_v$ is split into two cells $\square_w$ and $\square_z$ by an axis-parallel hyperplane h_v such that the interior of each cell contains at most $|P_v|/2$ points of P_v. If the depth of v is i, then h_v is parallel to the $((i \bmod d) + 1)$-th axis. The cell $\square_w$ (resp. $\square_z$) is associated with the child w (resp. z) of v. Since the partitions are balanced, the tree has height $O(\log n)$. The size of the tree is $O(n)$. It is well known that the tree can be constructed in $O(n \log n)$ time by first sorting P along each axis and then constructing it level by level in a top-down manner, spending $O(n)$ time at each level. See [68] for details.

Queries are performed recursively in a top-down manner starting with the root node. Given a query rectangle ρ and a node v of the tree, there are three cases : (a) $\square_v \subseteq \rho$, then all points of P_v are reported , (b) $\square_v \cap \rho = \varnothing$, there is nothing to be done, (c) $\square_v \cap \partial\rho \neq \varnothing$, we recurse on the children of v.

2.3.1 An MPC algorithm

We now describe an algorithm for constructing the kd-tree, denoted by $\mathsf{T} := \mathsf{T}(P)$, of linear size that has $O(s^{1-1/d} + k)$ query time. Our algorithm uses only $O(1)$ rounds of computation to build the tree. T is constructed and stored recursively in a distributed fashion across all machines. If $|P| = O(s)$, T is stored on a single machine and constructed using the sequential algorithm mentioned above. Otherwise we choose a parameter r, and the top subtree T^r of T containing $\log_2 r$ levels and $\Theta(r)$ leaves is built and stored on one machine, say β. The leaves of T^r partition P into subsets, each subset P_i associated with leaf l_i residing on its own consecutive set of machines I_i (the sets of machines being pairwise disjoint). For each such set P_i, $\mathsf{T}(P_i)$ is built and stored recursively using the machines in I_i. A machine may store multiple subtrees, one from each level of the recursion. However, the space used on

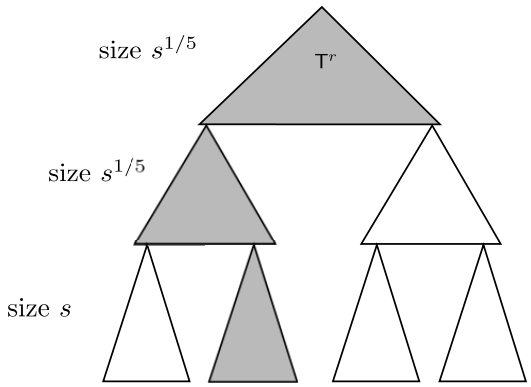


FIGURE 2.4: T^r is stored on a single machine. The leaves of T^r partition P into smaller subsets, each represented by their own recursively constructed kd-trees. Shaded subtrees may be stored on a single machine; however the space used on a single machine is still $O(s)$.

a single machine is still $O(s)$. See Figure 2.4.

The key to our $O(1)$ round algorithm is the use of an ε-approximation to build the top subtree T^r. While the size of all $P_i's$, the subsets associated with the leaves of T^r, will not be *exactly* the same, as for the standard kd-tree, the usage of ε-approximation will ensure that $|P_i| \leqslant 2|P|/r$, which in turn will guarantee that the height of T is at most $\log_2 n + O(1)$.

We now describe the recursive procedure `Build-kd-tree`$(S, \square, I)$, which for a rectangle $\square$, builds the tree $T(S)$ on $S = P \cap \square$ using a contiguous subset I of $\frac{|S|}{n}m$ machines, say $\{\beta, \beta+1, \ldots, \beta + \frac{|S|}{n}m - 1\}$. If $|I| = 1$, i.e., $|S| = O(s)$, then $\mathsf{T}(S)$ is constructed on machine β using the sequential algorithm. So assume $|I| > 1$.

Set $r = s^{1/5}$. Let $\Sigma = (S, \mathcal{R})$ be the range space where the ranges are induced by rectangles, i.e., $\mathcal{R} = \{S \cap \square \mid \square \text{ is a rectangle}\}$. We compute a $(1/r)$-approximation R of Σ of size $cr^2 \ln n$, for some constant $c > 0$, by calling the (randomized) procedure `Sample`(S, r, β). We compute on machine β the top subtree T^r of the standard kd-tree on R so that $|\mathsf{R}_v| \leqslant cr \ln n$ for every leaf v of T^r. The height of T^r is $\log_2 r$. Let `Partial-kd-tree`(R, r) denote this procedure. We assume this procedure returns

the tree T^r as well as the partition Π' of $\square$ induced by the rectangles associated with the leaves of T^r. By calling $\texttt{Partition}(S, I, \Pi', \beta)$, we partition S so that for each rectangle $\square_v$ of Π', $S_v = S \cap \square_v$ lies in a contiguous subset I_v of machines of I; $|I_v| = \frac{|S_v|}{n} \cdot m$. If $|S_v| > \frac{2|S|}{r}$ for some leaf v of T^r, we discard R and T^r, and repeat the above step. Otherwise, we recursively compute $(S_v, \square_v, I_v)$ for all leaves v of T^r. Algorithm 1 describes the pseudocode.

Algorithm 1 $\texttt{Build-kd-tree}(S, \square, I)$

1: If $|I| = 1$, compute $\mathsf{T}(S)$ sequentially.
2: $r = s^{1/5}$.
3: $\mathsf{R} \leftarrow \texttt{Sample}(S, r, \beta)$.
4: $\mathsf{T}^r, \Pi' \leftarrow \texttt{Partial-kd-tree}(\mathsf{R}, r)$.
5: $\texttt{Partition}(S, I, \Pi', \beta)$.
6: **for all** $\square_v \in \Pi'$ in parallel **do**
7: $\quad$ $\texttt{Build-kd-tree}(S_v, \square_v, I_v)$
8: **end for**

Lemma 7. *T^r has $\Theta(r)$ leaves. If R is a $(1/r)$-approximation of Σ then $|S_v| \leqslant \frac{2|S|}{r}$ for all leaves of T^r.*

Proof. Since the depth of T^r is $\log_2 r$, it has $\Theta(r)$ leaves. For any leaf $v \in \mathsf{T}^r$, let $\mathsf{R}_v = \mathsf{R} \cap \square_v$. If R is a $(1/r)$-approximation of Σ, then

$$\left| \frac{|S_v|}{|S|} - \frac{|\mathsf{R}_v|}{|\mathsf{R}|} \right| \leqslant \frac{1}{r}.$$

Therefore,

$$|S_v| \leqslant |S| \left(\frac{1}{r} + \frac{|\mathsf{R}_v|}{|\mathsf{R}|} \right) \leqslant |S| \left(\frac{1}{r} + \frac{cr \ln n}{cr^2 \ln n} \right) = \frac{2|S|}{r}.$$

□

Since R is a $(1/r)$-approximation with probability at least $1 - 1/n^{\Omega(1)}$, the algorithm succeeds in one attempt with probability at least $1 - 1/n^{\Omega(1)}$. The depth of recursion is $O(\log_r n)$. Since $r = s^{1/5}$ and we assume $s = n^\alpha$ for some constant $\alpha > 0$,

the depth of recursion is $O(1)$. By Lemmas 4 and 6 (i), the running time of the algorithm is $O(s \log s)$ and the total work performed is $O(n \log n)$ with probabillity at least $1 - 1/n^{\Omega(1)}$.

Lemma 8. *The height of the tree constructed by the algorithm is at most* $\log_2 n + O(1)$.

Proof. The base case, i.e. $|I| = 1$, constructs a subtree of height $\log_2 s$. `Partial-kd-tree` constructs a subtree of height at most $\log_2 r$. Since the depth of the recursion is at most $\log_{r/2}\left(\frac{n}{s}\right)$, the total height of the tree is at most

$$\begin{aligned}
&\log_2 s + \log_{r/2}\left(\frac{n}{s}\right) \cdot \log_2 r \\
&\leqslant \log_2 s + \log_{r/2}\left(\frac{n}{s}\right)\left(1 + \log_2\left(\frac{r}{2}\right)\right) \\
&\leqslant \log_2 s + \log_2\left(\frac{n}{s}\right) + \left(\frac{\log_2(n/s)}{\log_2 r - 1}\right) \\
&\leqslant \log_2 n + \frac{(1-\alpha)\log_2 n}{(\alpha/5)\log_2 n - 1} \\
&= \log_2 n + O(1).
\end{aligned}$$

□

Alternatively, the deterministic version of $\texttt{Sample}(S, r, \beta)$ can be used to construct a $(1/r)$-approximation R. Since the deterministic procedure is more expensive, we choose $r = s^\beta$ where $\beta \leqslant 1/5$ is a sufficiently small constant. The depth of recursion remains $O(\log_r n) = O(1/\beta)$. By Lemma 6 (ii), the running time of the algorithm is $s^{1+O(\beta)}$ and the total work peformed by the algorithm is $n^{1+O(\beta)}$.

2.3.2 Query procedure

Given a query rectangle ρ and the set of machines I containing T, in the first round we perform the query locally on the machine β containing the top subtree T^r. This

gives us a set of leaves of T^r whose squares intersect with ρ. For each such leaf v, the query is performed recursively in parallel on the subtree rooted at v contained in the machines I_v.

The number of levels of recursion is the same as that of the construction algorithm, i.e., $O(1)$. This is also the number of rounds of computation required. The following lemma bounds the query procedure's work.

Lemma 9. *The total work performed by the query procedure is $O(n^{1-1/d} + k)$ where k is the number of points in the query rectangle.*

Proof. For simplicity, we prove the lemma for $d = 2$; the proof is similar for higher values of d. Let ρ be a query rectangle. The total work performed by the query procedure is $O(k)$ plus the number of nodes v in$\mathsf{T}$ such that $\partial\rho$ intersects $\square_v$. Fix an edge e of ρ. Since the splitting line alternates between being horizontal and vertical, e intersects the cells associated with at most two grandchildren of a node v; see [68]. Let $\varphi(h)$ denote the number of nodes in a subtree of height h that intersect e. We obtain the following recurrence:

$$\varphi(h) \leqslant 2\varphi(h-2) + 3.$$

The solution to the above recurrence is $\varphi(h) = O(2^{h/2})$. By Lemma 8, the height of T is at most $\log_2 n + O(1)$. We obtain that e intersects the cells associated with $O(\sqrt{n})$ nodes of T. Hence, $\partial\rho$ intersects $O(\sqrt{n})$ cells of T. This completes the proof of the lemma. □

The time required by the query procedure can be similarly bounded by $O(s^{1-1/d} + k')$, where k' is the maximum number of points reported by a single machine. We thus have the following.

Theorem 10. *Let P be a set of n points in $\mathbb{R}^d$. A kd-tree on P can be constructed in the MPC model so that an orthogonal range-reporting query can be answered using*

$O(1)$ rounds of computation and $O(n^{1-1/d} + k)$ work, where k is the output size; the running time is $O(s^{1-1/d} + k')$ where k' is the maximum number of points reported by a machine. The tree can be built in $O(1)$ rounds, $O(n \log n)$ work, and $O(s \log s)$ time with probability $1 - 1/n^{\Omega(1)}$. Alternatively, for a parameter $\beta \leqslant 1/5$, it can be constructed deterministically in $O(1/\beta)$ rounds, $s^{1+O(\beta)}$ time, and $n^{1+O(\beta)}$ work.

2.3.3 Extension to simplex range searching

Given a set of n points P in $\mathbb{R}^d$, Chan [58] described a partition tree that can answer simplex range-reporting queries (*i.e.,* reporting points lying in a simplex) in $O(n^{1-1/d} + k)$ time using $O(n)$ space. The expected time to construct the partition tree is $O(n \log n)$. Like *kd*-trees, partition trees represent a hierarcical decomposition of space into cells; however each cell is a simplex. The number of points within a cell decreases by a constant factor at every level, so the tree has height $O(\log n)$. Any arbitrary hyperplane intersects at most $O(n^{1-1/d})$ cells in the tree, hence a simplicial range-reporting query can be answered in $O(n^{1-1/d} + k)$ time, where k is the number of points reported.

Our algorithm for constructing a *kd*-tree can be extended to build a partition tree in the MPC model in $O(1)$ rounds of computation. Omitting all the details, we conclude the following:

Theorem 11. *Let P be a set of n points in $\mathbb{R}^d$. A partition tree on P can be constructed in the MPC model so that a simplicial range-reporting query can be answered using $O(1)$ rounds of computation and $O(n^{1-1/d} + k)$ work, where k is the output size; the running time is $O(s^{1-1/d} + k')$ where k' is the maximum number of points reported by a machine. The tree can be built in $O(1)$ rounds in expected time $O(s \log s)$ and $O(n \log n)$ expected work. Alternatively, for a parameter $\beta \leqslant 1/5$, it can be constructed deterministically in $O(1/\beta)$ rounds, $s^{1+O(\beta)}$ time, and $n^{1+O(\beta)}$ work.*

2.4 NN Searching and BBD-tree

Given a set of points P and a query point q in $\mathbb{R}^d$, a $(1+\varepsilon)$-nearest neighbor (ε-NN) of q is a point in P whose distance from q is within a factor of $(1+\varepsilon)$ of the distance between q and its closest point in P. The goal is to process P into a data structure so that for a query point q, an ε-NN of q in P can be reported quickly. Arya et al. [34] proposed the *balanced-box decomposition* (BBD) tree for answering ε-NN queries. This section describes building a BBD-tree in the MPC model.

Given n points $P \in \mathbb{R}^d$, the BBD-tree of P, denoted by $\mathsf{B}(P) := \mathsf{B}$, is a binary tree of height $O(\log n)$. A node $v \in \mathsf{B}$ stores a cell $\square_v$ and a representative point $p_{\square_v} \in P$ lying inside $\square_v$. The cell $\square_v$ is either a d-dimensional rectangle or the region between two nested rectangles. All rectangles have *aspect ratio* (the ratio between the longest and the shortest side) bounded by 3. The root cell is large enough to contain the entire set P. The cell $\square_v$ (with possibly a hole inside) is split into two cells $\square_w$ and $\square_z$ in one of two ways:

(a) by an axis-parallel hyperplane not intersecting the hole (if any), or

(b) by an axis-parallel rectangle containing the hole (if any).

See Figure 2.5. The cell $\square_w$ (resp. $\square_z$) is associated with the child w (resp. z) of v. Like a *kd*-tree, a BBD-tree induces a hierarchical partition of $\mathbb{R}^d$. The *size* of a cell is the length of its longest side. Arya et al. show that cells at each node can be split in a way so that the number of points inside the cells reduces by at least a factor of 2/3 every 4 levels of B, and the size of cells decreases by at least a factor of 2/3 every $4d$ levels of the tree. Each leaf cell contains at most one input point. The crucial observation is that the range space induced by the cells in a BBD-tree has constant VC-dimension (see Section 2.2.1), and we can adapt the *kd*-tree construction algorithm for building the BBD-tree. The only difference is that

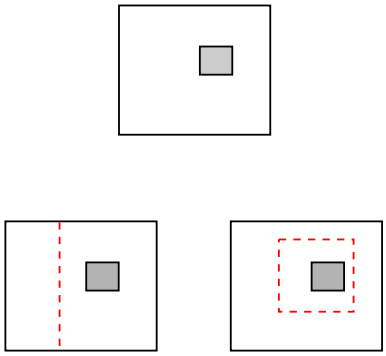


FIGURE 2.5: Two ways to divide a cell (with one hole) into two cells in a BBD-tree.

instead of building a *kd*-tree locally in the procedure `Partial-kd-tree`, we build a local BBD-tree. The tree is also stored in a manner similar to the *kd*-tree, i.e., $\mathsf{B}(P)$ is stored in a distributed fashion across all machines.

Query procedure. We first describe a sequential query procedure that is slightly different from the one in [34], and then show how to implement it in the MPC model.

Given a query point q, the query procedure proceeds top-down, level by level. At each step, it maintains an estimate r_{curr} of distance from q to its nearest neighbor, and it stores the set of active nodes in a queue Q. Algorithm 2 summarizes the query procedure.

Algorithm 2 `NN-Query`(q)

$r_{curr} \leftarrow ||qp_{root}||$, $p_{curr} \leftarrow p_{root}$.
$Q \leftarrow \{root\}$.
while $Q \neq \varnothing$ **do**
    $v \leftarrow$ `Dequeue`(Q).
    if $||qp_v|| < r_{curr}$ **then**
        $r_{curr} \leftarrow ||qp_v||$, $p_{curr} \leftarrow p_v$.
    end if
    if $\mathtt{d}(q, \square_v) < \frac{r_{curr}}{1+\varepsilon}$ and v not a leaf **then**
        `Enqueue`(w, Q) for all child w of v.
    end if
end while
Return p_{curr}.

Lemma 12. *The above query procedure returns an ε-NN of q.*

Proof. Let p^* be the actual nearest neighbor of q. We show that at the end of the procedure $r_{curr} \leqslant (1+\varepsilon)||qp^*||$.

Note that the value of r_{curr} is non-increasing throughout the procedure. Let v be the last node examined by the procedure such that $p^* \in \square_v$. If v is a leaf, then $r_{curr} = ||qp^*||$. Otherwise, v was discarded, in which case $\mathtt{d}(q, \square_v) \geqslant \dfrac{r_{curr}}{1+\varepsilon}$. However, $||qp^*|| \geqslant \mathtt{d}(q, \square_v)$ and hence $r_{curr} \leqslant (1+\varepsilon)||qp^*||$. □

We use the following fact, proved as Lemma 4 in [34], to bound the number of cells examined at each level.

Fact 13. *Given a BBD-tree for a set of data points in $\mathbb{R}^d$, the number of cells of size at least $\Delta > 0$ that intersect a ball of radius r is at most $\lceil 1 + 6r/\Delta \rceil^d$.*

The next lemma is then a simple variant of Lemma 5 in [34].

Lemma 14. *The query procedure visits at most $\lceil 1 + 6d/\varepsilon \rceil^d$ cells at any level.*

The MPC implementation of the query procedure is obtained by modifying Algorithm 2 as follows. Let q be a query point, and let I be the set of machines that store B. The first round runs Algorithm 2 on the machine β containing the top subtree B^r, having $O(\log r)$ levels. When the procedure finishes traversing B^r, by Lemma 14, Q has $O(\frac{1}{\varepsilon^d})$ leaves of B^r. For each such leaf v, we pass the values p_{curr} and r_{curr} to the machines I_v containing the subtree B_v rooted at v, and Algorithm 2 is then run recursively in parallel on all B_v for $v \in Q$. At the end, we return the point that is nearest to q among all the points returned by these subtrees.

The query procedure performs $O(1)$ rounds of computation. The number of recursive subproblems increases by a factor of $O\left(\frac{1}{\varepsilon^d}\right)$ at every level of recursion. The subproblems at each level of recursion run in parallel and take time $O\left(\frac{1}{\varepsilon^d} \log\left(\frac{1}{\varepsilon}\right) \log n\right)$. Since the number of recursive subproblems is $\frac{1}{\varepsilon^{O(d)}}$, the total work done is $\frac{1}{\varepsilon^{O(d)}} \log n$. We thus have the following.

Theorem 15. *Given a set P of n points in $\mathbb{R}^d$, a BBD-tree on P can be built in the MPC model that can answer $(1+\varepsilon)$-nearest neighbor queries in $O(1)$ computation rounds, $O\left(\frac{1}{\varepsilon^d}\log\left(\frac{1}{\varepsilon}\right)\log n\right)$ time and $\frac{1}{\varepsilon^{O(d)}}\log n$ work, for some constant $c \geqslant 2$. The tree can be built in $O(1)$ rounds of computation, $O(n\log n)$ work, and $O(s\log s)$ time with probabillity $1-1/n^{\Omega(1)}$. Alternatively, for a parameter $\beta \leqslant 1/5$, it can be constructed deterministically in $O(1/\beta)$ rounds, $s^{1+O(\beta)}$ time, and $n^{1+O(\beta)}$ work.*

2.5 Range Tree

Given a set P of n points in $\mathbb{R}^d$, a d-dimensional range tree on P, denoted by $\mathsf{T} := \mathsf{T}(P)$, can answer orthogonal range queries in $O(\log^{d-1} n + k)$ time using $O(n\log^{d-1} n)$ space, where k is the number of points reported [68]. T is defined recursively. For $d = 1$, T is a sorted array or a balanced binary search tree. For $d > 1$, T consists of a primary tree $\mathsf{T}_0 := \mathsf{T}_0(P)$, and each node of T_0 stores a $(d-1)$-dimensional range tree as a secondary structure. T_0 is a balanced binary search tree on the x_1-coordinates of P. Each node $v \in \mathsf{T}_0$ stores an interval $\mathcal{I}_v$. Let P_v be the points whose x_1-coordinates lie in $\mathcal{I}_v$ (the root has the set P and the interval spanning all of P). The interval $\mathcal{I}_v$ is split into $\mathcal{I}_w$ and $\mathcal{I}_z$ so that $|P_w|, |P_z| \leqslant \lceil |P_v|/2 \rceil$, where w and z are the children of v. Let $P_v^{\perp}$ denote the projection of P_v onto the hyperplane $x_1 = 0$. Node v also has a secondary data structure $\mathsf{T}(P_v^{\perp})$, a $(d-1)$-dimensional range tree on $P_v^{\perp}$. A simple recursive argument shows that the size of T is $O(n\log^{d-1} n)$, and that it can be constructed in $O(n\log^{d-1} n)$ time after sorting P along each of its coordinates.

Let $\rho = [a_1, b_1] \times \ldots \times [a_d, b_d]$ be a query rectangle, and let $\rho^{\perp} = [a_2, b_2] \times \ldots \times [a_d, b_d]$. Let w_a (resp. w_b) be the leaf of the primary tree such that $a_1 \in \mathcal{I}_{w_a}$ (resp. $b_1 \in \mathcal{I}_{w_b}$), and let v^* be the lowest common ancestor of w_a and w_b. Let V_ρ be the set of nodes v such that either v is the right child of its parent and the left sibling of v lies on the path from v^* to w_a or v is the left child of its parent and its right sibling lies on the path from v^* to w_b. It is known [68] that $P \cap \rho = \bigcup_{v \in V_\rho}(P_v \cap \rho)$ and a

point $p \in P_v \cap \rho$, for $v \in V_\rho$, if and only if $p^\perp \in P_v^\perp \cap \rho^\perp$. Hence, we recursively query $P_v^\perp$ with $\rho^\perp$ for all $v \in V_\rho$.

2.5.1 An MPC algorithm

Since the total space required by T is $O(n \log^{d-1} n)$, we slightly amend our model so that the memory and I/O size for each machine per round is $O(s \log^{d-1} n)$. We still have m machines with $s = n/m$ and $s \geqslant n^\alpha$ for some positive constant $\alpha < 1$. For each $\ell \in \{1, \ldots, d\}$, let $s_\ell = s \log^{\ell-1} n$. We assume that the input points are initially distributed so that each machine contains $O(s_1) = O(s)$ points. This assumption can be guaranteed by redistributing the points using an algorithm similar to the `Partition` procedure described in Section 2.2.2. Our construction procedure recursively builds primary trees for each coordinate x_ℓ using $O(s_\ell)$ memory and I/O size per machine.

T is stored in a distributed fashion. Suppose we are building an ℓth level structure of the range tree (initially, $\ell = 1$). The primary tree T_0 for this level is built and stored in a manner similar to the kd-tree, using a procedure we call `Build-primary-tree`(P, I, ℓ), which is nearly the same as the one used to build the kd-tree. The only difference is that the range space is induced by intervals over the real line, whose VC-dimension is a constant. Briefly, if $|P| = O(s_\ell)$, then T_0 is stored on a single machine and constructed using the sequential algorithm. Otherwise, we choose a parameter r, and the top subtree T_0^r of T_0 containing $\log_2 r$ levels and $\Theta(r)$ leaves is built and stored on a machine β. The leaves of T_0^r again partition P, and the subtrees of T_0 rooted at these leaves are built and stored recursively using disjoint sets of machines. We can also build T_0 deterministically in the same amount of time, work, and rounds by using a simpler deterministic `Sample` procedure since our ranges are just intervals over the real line. We omit details for the deterministic procedure.

Each node v of T_0 has a pointer to a secondary structure. Each bottom-most

subtree of T_0 of size $O(s_\ell)$ stored on a single machine has all the secondary structures of its nodes stored on the same machine. These secondary structures are built using the sequential algorithm in $O(s_\ell \log^{d-\ell} n)$ time and space each. The secondary structures for the remaining nodes are stored on disjoint subsets of I. See Figure 2.6.

To build the secondary structures, each point p is copied $\Theta(\log |P|)$ times, sending the copies to the disjoint sets of machines that will store $\mathsf{T}(P_v^{\perp})$ for all P_v containing p. We name our copying procedure `Copy-points`(P, I, T_0). Each machine of $I = \{i_0, i_0 + 1, \ldots\}$ contains several points lying in the leaves of T_0. To facilitate our copying procedure, we inform each machine about the nodes of T_0 that lie above its points. Let β be the machine storing T_0^r. We run the procedure `Broadcast`$(\mathsf{T}_0^r, I, \beta)$ and then recursively repeat the broadcast procedure with the child subtrees of T_0^r and their disjoint sets of machines. In $O(1)$ rounds, each machine will receive the ancestor subtrees for its points.

Let $\lambda = \Theta(\log |P|)$ be the depth of T_0. Intuitively, the set I is divided into λ equal-sized sets of size $|I|/\lambda$, one for each level of T_0. Sets P_v are then distributed among disjoint subsets of machines from those machines set aside for level λ. Now, consider a node v of T_0. Let λ_v be the depth of node v in T_0, $\{v_0, v_1, \ldots\}$ be the nodes of T_0 lying at depth λ_v, and $v_j = v$. Let $I_v = \{i_v, i_v + 1, \ldots,\}$ be the set of machines storing the points P_v. Let $p \in P_v$, and let i be the machine storing p. Finally, let c be a sufficiently small constant. Machine i sends a copy of p to machine $i_0 + \lambda_v \cdot cm/\lambda + cmj/(2^{\lambda_v}\lambda) + \lfloor c(i - i_v)/\lambda \rfloor - 1$. Let I'_v be the set of machines that receive the points of P_v, the set that is used to construct the secondary structure for v. Each point is copied $\Theta(\log |P|)$ times, so the total communication out of i while copying points is $O(s_\ell \log |P|)$.

Our construction algorithm concludes by recursively building the secondary structures (at the $(\ell+1)$st level) for each node v on the set of machines I'_v. We describe our construction procedure in Algorithm 3. The procedure `Build-range-tree` takes ℓ

as one of its parameters to account for the storage required when building each level of the data structure.

Algorithm 3 `Build-range-tree`(P, I, ℓ)

1: If $|I| = 1$, build $\mathsf{T}(P)$ sequentially.
2: $\mathsf{T}_0(P) \leftarrow$ `Build-primary-tree`(P, I, ℓ).
3: `Copy-points`(P, I, T_0).
4: **for all** $v \in \mathsf{T}_0$ **do**
5: `Build-range-tree`$(P, I'_v, l + 1)$
6: **end for**

Lemma 16. *Algorithm* `Build-range-tree`$(P, I, 1)$ *takes* $O(1)$ *rounds of computation,* $O(n \log^d n)$ *work, and* $O(s \log^d n)$ *time.*

Proof. Consider running `Build-range-tree`(P, I, ℓ) for an arbitrary $\ell \in \{1, \ldots, d\}$. Step 1 can be done locally in $O(s_\ell \log^{d-\ell} n)$ time. Step 2 can be done in $O(1)$ rounds using $O(s_\ell \log n)$ time and $O(m \cdot s_\ell \log n)$ work. Copying points can also be done in one round using $O(s_\ell \log n)$ time and $O(m \cdot s_\ell \log n)$ work. There are $O(1)$ levels in the data structure, and each is built in $O(1)$ rounds, so the entire construction procedure takes $O(1)$ rounds of computation. The running time $T(\ell)$ for building ℓth and lower levels of a d-dimensional range tree with $O(s_\ell)$ points initially stored on each machine can be expressed using the recurrence $T(\ell) = O(s_\ell \log n) + T(\ell + 1)$ with a base case of $T(d) = O(s_d \log n) = O(s \log^d n)$. This recurrence solves to $T(\ell) = O(s \log^d n)$. Similarly, the total work is $O(n \log^d n)$. □

2.5.2 Query procedure

Given a query ρ, we run it through the primary structure T_0 to get the $O(\log n)$ nodes whose secondary structures have to be probed further (as described in the beginning of Section 2.5). This takes $O(\log n)$ time and work and $O(1)$ rounds. Each secondary structure is then probed recursively.

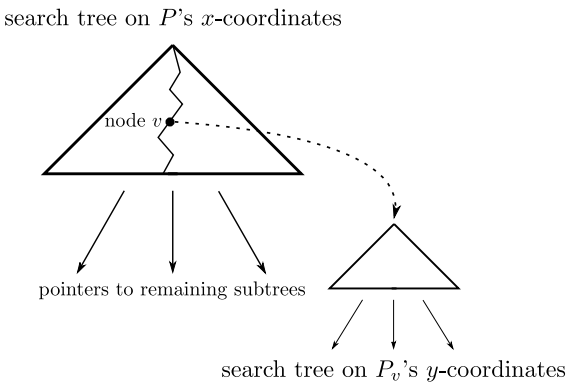


FIGURE 2.6: A distributed range tree in 2 dimensions. A balanced binary search tree over the x-coordinates is stored in a recursive manner. Each node points to a distributed binary search tree over the y-coordinates.

In total, we take $O(\log n)$ time and $O(\log^{d-1} n)$ work over $O(1)$ rounds finding secondary structures to probe that are stored in the distributed manner described above. The actual bottleneck is the $O(\log^d n + k')$ time and $O(\log^d n + k)$ work required to query the bottom-most subtrees stored entirely within individual machines, where k' and k are the maximum points reported by a machine and the total number of points reported respectively.

We thus have the following theorem.

Theorem 17. *Given a set P of n points in $\mathbb{R}^d$, a range tree $\mathsf{T}(P)$ on P of size $O(n \log^{d-1} n)$ can be constructed that can answer an orthogonal range-reporting query in $O(\log^d n + k')$ time and $O(\log^d n + k)$ work in the MPC model in $O(1)$ rounds of computation, where k' is the maximum number of points reported by a machine and k is the total number of points reported. The tree can be built in $O(1)$ rounds, $O(n \log^d n)$ work, and $O(s \log^d s)$ time.*

2.6 Conclusion

We presented efficient algorithms to build and query *kd*-trees, range trees, and BBD-trees in the MPC model. Our algorithms were based on recursively partitioning a

set of points in $\mathbb{R}^d$ by first sampling a small set of points and then computing the partition based on the sample. We believe our framework may be useful in designing efficient data structures in other domains or in the direct analysis of data.

We leave several questions open for future research. The one most closely related to our current work is whether our algorithms for *kd*-trees and BBD-trees can be made deterministic while remaining work-optimal. Our algorithms use $\text{polylog}_s\, n = O(1)$ rounds; it will be interesting to see if this can be improved to $O(\log_s n)$. A lower bound result from [89] says that $\Omega(\log_s n)$ rounds are needed to construct the data structures. Further afield, we ask what other data structures can be constructed efficiently in massively parallel models like MPC; can we efficiently build geometric data structures with more direct GIS applications such as those supporting fast point location queries? Can we efficiently parallelize geometric and topological data analysis methods such as persistent homology (see [40, 39, 108] for parallel and distributed algorithms for persistent homology, however these do not have any provable performance guarantees)? Finally, can our techniques be used outside the geometric domain? In particular, it would be interesting to see if similar hierarchical data structures can be used to efficiently answer graph connectivity and related queries after some preprocessing.

3

Analyzing Massive Terrains

3.1 Introduction

The problem of modeling and analyzing massive terrain data sets has been studied extensively in many disciplines such as GIS, spatial databases, computational geometry, and environmental sciences. Formally, a terrain Σ is an xy-monotone surface in $\mathbb{R}^3$ and can be viewed as the graph of a height function $h : \mathbb{R}^2 \to \mathbb{R}$. It is often stored using a digital elevation model (DEM) such as a triangulated xy-monotone surface known as a *triangulated irregular network* (TIN). Here, the surface Σ can be regarded as a triangulation $\mathbb{M}$ of $\mathbb{R}^2$ and a piecewise linear (within each triangle of $\mathbb{M}$) height function $h : \mathbb{M} \to \mathbb{R}$. Because of various nice properties, $\mathbb{M}$ is often the Delaunay triangulation of the points in the plane at which the height of the terrain is observed.

Suppose we have computed a suitable TIN using triangulation $\mathbb{M}$ and height function h. Given a height value ℓ, the ℓ-level set of h is the set of all points on $\mathbb{M}$ whose height values are ℓ. As one continuously varies ℓ, the level sets of h deform, and their topology changes at certain heights. The *contour tree* of h encodes the

changes to the levels sets. The contour tree is used in a variety of applications such as prediction of water flow including the computation of watershed hierarchies, the visualization of hydrothermal plumes, and the visualization of climate models [32, 42, 67, 117]. Therefore, it is natural to compute the contour tree of h as a second step to analyzing the terrain it represents.

Recent advances in sensing technology provide both opportunities and obstacles for the aforementioned analysis of terrain data. On the one hand, data like the point sets created by LiDAR are being generated at increasingly higher resolutions, expanding the types and quality of data analyses that are even possible. On the other hand, classical algorithms that run on individual machines cannot keep up with the amount of data generated.

In this chapter, we study the problems of constructing the TIN DEM of such large elevation data (e.g., LiDAR data sets), a set of points in $\mathbb{R}^3$, and then constructing the contour tree of this TIN DEM. We use the Massively Parallel Communication (MPC) model of computation, discussed in more detail in Chapter 2.

3.1.1 Related work

Computing the Delaunay triangulation (and its dual, the Voronoi diagram) is one of the most studied problems in computational geometry; see [35]. Many of the algorithms for computing the Delaunay triangulation in the RAM model run in $O(n \log n)$ time. There are algorithms for computing the Delaunay triangulation in the I/O-model [65] and the PRAM model (see [45] and the references therein). In addition, Goodrich [88] describes an algorithm for a bulk-synchronous parallel (BSP) computer that constructs convex hulls in 3D, which can be adapted to construct Delaunay triangulations in $O(1)$ rounds in the MPC model of computation, although the algorithm itself is rather involved. SpatialHadoop [79] includes a MapReduce algorithm to compute the Delaunay triangulation of a point set, though there are no

provable bounds on its time-complexity. Finally, there exist efficient algorithms for constructing the *constrained* Delaunay triangulation which is required to contain a given subset of edges [35]; however, these algorithms are not for the MPC model.

Efficient algorithms for constructing contour trees were discovered more recently. Van Kreveld *et al.* [142] gave the first $O(n \log n)$ time algorithm for our setting of piecewise linear height functions over $\mathbb{R}^2$. This algorithm was later generalized by Tarasov and Vyalyi [139] and Carr *et al.* [56]. Agarwal *et al.* [15] gave an I/O-efficient algorithm for computing the contour tree. There has been a great deal of work on computing contour trees and similar structures using parallel and distributed computing, although these algorithms do not have any theoretical guarantees for the MPC model. See [9, 122, 123] and the references therein. Of particular relevance to our work are the distributed contour and *merge* trees of Morozov and Weber [122, 123]. Suppose one has a TIN $\mathbb{M}$ and height function h stored in a distributed manner. Morozov and Weber describe how to compute trees for the data local to each processor, and then merge simplified versions of these trees so that the simplification is known to all of the processors.

3.1.2 Our results

In this chapter, we describe new algorithms in the MPC model for both parts of the terrain analysis pipeline mentioned above. Given a point set P in $\mathbb{R}^2$ with $|P| = n$, our first result is a randomized algorithm that computes the Delaunay triangulation of P using, with high probability, $O(1)$ rounds of computation in $O(s \log^2 n)$ time and $O(n \log n)$ work. While Goodrich's [88] convex hull algorithm for BSP computers can be adapted for a similar result, our approach is more direct, and is considerably simpler.

Our second result is a deterministic algorithm that computes the contour tree for a height function h over triangulation $\mathbb{M}$ using $O(1)$ rounds of computation,

$O(s \log n)$ time, and $O(n \log n)$ work, provided that the following two assumptions hold:

1. $s = \Omega(n^{1/2+\varepsilon})$ for some constant $\varepsilon > 0$, and
2. every line in $\mathbb{R}^2$ intersects $O(\sqrt{n})$ edges of $\mathbb{M}$.

We believe these are reasonable assumptions, because the number of bytes of memory available to individual machines in datacenters should be much higher than the number of available machines (justifying the first assumption). Our second assumption holds for sure when $\mathbb{M}$ is the Delaunay triangulation of points lying on a grid, and it holds with probability at least $1 - 1/n^{\Omega(1)}$, for n sufficiently large when the points are chosen independently and uniformally at random from the unit square [46]. Furthermore, large data sets generated by LiDAR and other technologies have points almost uniformly sampled.

Both of our algorithms rely on a similar divide-and-conquer strategy. Namely, we distribute the input points (triangles) into subsets with small intersection. These subsets are then copied to disjoint sets of machines where we recursively compute the Delaunay triangulation (contour tree) for each subset independently and in parallel. For our Delaunay triangulation algorithm, we begin by sampling a small subset of points S and building the Delaunay triangulation of the sample. The set of input points are then distributed based on how they conflict with edges in the triangulation of S. This idea was used before in, for example, the constrained Delaunay triangulation algorithm of Agarwal *et al.* [14], but implementing the scheme in the MPC model requires additional ideas. We need high probability bounds instead of simple expectation bounds on the size of "conflict lists", and this involves adapting the *exponential decay lemma* [115] to our scenario using a two-level sampling procedure.

For our algorithm to build contour trees, we combine recursively constructed contour trees for each subset of triangles, taking advantage of the subsets' small in-

tersection. Unlike the algorithm of Morozov and Weber [122, 123], we do not rely on an initial distribution of the input triangles to induce small regions of the triangulation with limited intersection. Instead, we compute our own recursive hierarchy of triangle subsets that contain few boundary vertices. Further, we combine multiple trees simultaneously to minimize communication instead of combining them in pairs.

3.2 Preliminaries

3.2.1 Delaunay triangulations

Given a set of n points $P \in \mathbb{R}^2$, a triangulation of P is a maximal subdivision of the plane into triangles having vertices from P. A triangle belonging to a given triangulation is said to be *Delaunay* if its circumcircle does not contain any point of P in its interior. A triangulation of P is called a *Delaunay triangulation*, denoted by $\mathcal{DT}(P)$, if all the triangles in $\mathcal{DT}(P)$ are Delaunay.

Given a planar point set P, the *Voronoi cell* of a point $p \in P$, $\mathrm{Vor}_P(p)$, is the set of all those points $q \in \mathbb{R}^2$ that are closer to p than any other point of P, i.e.,

$$\mathrm{Vor}_P(p) = \{q \in \mathbb{R}^2 \mid d(p,q) \leqslant d(p',q)\ \forall p' \in P\}.$$

The *Voronoi diagram* of P, $\mathrm{Vor}(P)$, is the subdivision of the plane into the Voronoi cells of the points of P. A classical result states that $\mathcal{DT}(P)$ and $\mathrm{Vor}(P)$ are *duals* of each other, i.e., there exists an edge between two points $p, q \in P$ in $\mathcal{DT}(P)$ iff $\mathrm{Vor}_P(p)$ and $\mathrm{Vor}_P(q)$ share an edge in $\mathrm{Vor}(P)$. See Figure 3.1.

3.2.2 Terrains and contour trees

Terrains. Let $\mathbb{M} = (V, E, F)$ be a triangulation of $\mathbb{R}^2$, with vertex, edge, and face (triangle) sets V, E, and F, respectively, and let $n = |V|$. We assume that $\mathbb{M}$ is finite and contains one boundary component (our algorithm can be modified easily to remove the boundary by assuming V contains a vertex v_∞ at infinity). Let $h : \mathbb{M} \to \mathbb{R}$

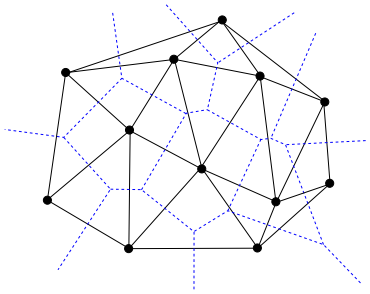
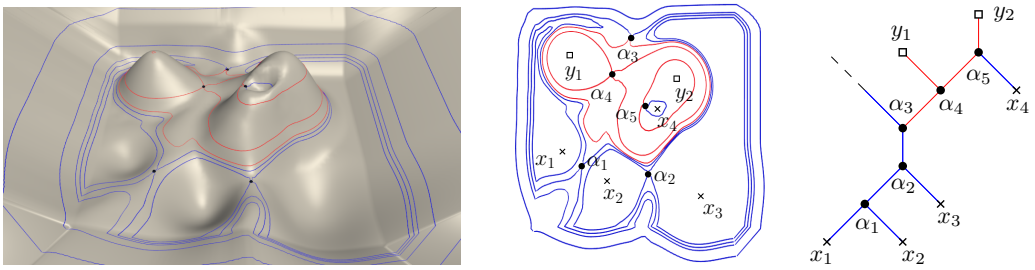


FIGURE 3.1: The Delaunay triangulation and Voronoi diagram (in blue dashed edges) of a set of points.



(a) (b) (c)

FIGURE 3.2: (a) (b) An example terrain depicted with contours through saddle vertices and showing the critical vertices of the terrain: $\alpha_1, \alpha_3, \alpha_4, \alpha_5$ are saddles. (c) The contour tree of the terrain in (a). This figure is taken from [22].

be a *height function*. We assume that the restriction of h to each triangle of $\mathbb{M}$ is a linear map, and that the heights of all vertices are distinct. Given $\mathbb{M}$ and h, the graph of h, called a *terrain* and denoted by Σ_h, is an xy-monotone triangulated surface whose triangulation is induced by $\mathbb{M}$. See Figure 3.2. If h is clear from context, we denote Σ_h as Σ. The vertices, edges, and faces of Σ are in one-to-one correspondence with $\mathbb{M}$. With a slight abuse of terminology, we refer to V, E, and F as vertices, edges, and triangles of both Σ and $\mathbb{M}$.

Critical points. For a vertex v of $\mathbb{M}$, the *link* of v, denoted $\mathrm{Lk}(v)$, is the cycle or path formed by the edges of $\mathbb{M}$ that are not incident on v but belong to the triangles incident to v. The lower (resp. upper) link of v, $\mathrm{Lk}^-(v)$ (resp. $\mathrm{Lk}^+(v)$), is the

subgraph of $\mathrm{Lk}(v)$ induced by vertices u with $h(u) < h(v)$ (resp. $h(u) > h(v)$). A *minimum* (resp. *maximum*) of $\mathbb{M}$ is a vertex v for which $\mathrm{Lk}^-(v)$ (resp. $\mathrm{Lk}^+(v)$) is empty. A maximum or a minimum vertex is called an *extremal* vertex. A non-extremal vertex v is *regular* if $\mathrm{Lk}^-(v)$ (and also $\mathrm{Lk}^+(v)$) is connected, and *saddle* otherwise. A vertex that is not regular is called a *critical* vertex.

Level sets and contours. Given any value $\ell \in \mathbb{R}$, the *ℓ-level set*, the *ℓ-sublevel set*, and the *ℓ-superlevel set* of $\mathbb{M}$, denoted as $\mathbb{M}_\ell$, $\mathbb{M}_{\leqslant \ell}$, and $\mathbb{M}_{\geqslant \ell}$, respectively, consist of points $x \in \mathbb{R}^2$ with $h(x) = \ell$, $h(x) \leqslant \ell$, $h(x) \geqslant \ell$, respectively. A connected component of $\mathbb{M}_\ell$ is called a *contour*. See Figure 3.2. Each point $v \in \mathbb{R}^2$ is contained in exactly one contour in $\mathbb{M}_{h(v)}$, which we call *the contour of v*. The contour of a noncritical point is a simple polygonal cycle with non-empty interior or a polygonal boundary-to-boundary arc. The contour of a local minimum or maximum v only consists of the single point v, and the contour of a saddle vertex v consists of two or more simple cycles and/or arcs with v being their only intersection point.

Contour trees. Consider raising ℓ from $-\infty$ to ∞. The contours continuously deform, but no changes happen to the topology of the level set as long as ℓ varies between two consecutive critical levels. A new contour appears as a single point at a minimum vertex, and an existing contour contracts into a single point and disappears at a maximum vertex. An existing contour splits into two new contours or two contours merge into one contour at a saddle vertex. The *contour tree* $\mathcal{T}_h$ of h is a tree on the critical vertices of $\mathbb{M}$ that encodes these topological changes of the level set. An edge (v, w) of $\mathcal{T}_h$ *represents* the contour that appears at v and disappears at w.

Formally, $\mathcal{T}_h$ is the quotient space in which each contour is represented by a point and connectivity is defined in terms of the quotient topology. Let $\rho : \mathbb{M} \to \mathcal{T}_h$ be the associated quotient map, which maps all points of a contour to a single point on an

edge of $\mathcal{T}_h$. Fix a point p in $\mathbb{M}$. If p is not a critical vertex, $\rho(p)$ lies in the relative interior of an edge in $\mathcal{T}_h$; if p is an extremal vertex, $\rho(p)$ is a leaf node of $\mathcal{T}_h$; and if p is a saddle vertex then $\rho(p)$ is a non-leaf node of $\mathcal{T}_h$. See Figure 3.2. We use h to denote the height function on the points of $\mathcal{T}_h$ as well. The definition of contour trees as a quotient map actually applies to any simply connected topological space Σ with associated height function h.

Join trees and split trees. Analogous to the contour tree of Σ which encodes the topological changes to $\mathbb{M}_\ell$ as we increase ℓ from $-\infty$ to ∞, the *join tree* $\mathcal{J}_h$ (resp. *split tree* $\mathcal{S}_h$) encodes the topological changes in $\mathbb{M}_{\leqslant \ell}$ (resp. $\mathbb{M}_{\geqslant \ell}$). Its leaves are minima (resp. maxima) of Σ, and its internal nodes are some of the saddle vertices of Σ. The contour tree, join tree, and split tree can all be *augmented* with additional vertices of $\mathbb{M}$ by subdividing the edges representing the contours, sublevel set components, or superlevel set components containing these vertices. Carr *et al.* [56] describe how to build the contour tree $\mathcal{T}_h$ in time linear in its size given the join and split trees augmented with each other's nodes.

3.2.3 ε-nets

We have already come across range spaces and VC dimension (see Section 2.2). The only ranges we consider in this chapter are those induced by the union of two open discs in $\mathbb{R}^2$. Such range spaces have constant VC dimension.

Given a range space $Y = (X, \mathcal{R})$ and $0 \leqslant \varepsilon \leqslant 1$, a subset $X' \subseteq X$ is called an *ε-net* of Y if for any range $R \in \mathcal{R}$ with $|R| \geqslant \varepsilon |X|$, set X' intersects R. Similarly, given a range space $Y = (X, \mathcal{R})$ and $0 \leqslant \varepsilon \leqslant 1$, a subset $X' \subseteq X$ is called an *ε-approximation* of Y if for any range $R \in \mathcal{R}$, we have

$$\left| \frac{|X' \cap R|}{|X'|} - \frac{|R|}{|X|} \right| \leqslant \varepsilon.$$

3.3 Computing Delaunay Triangulation

Let $P \subset \mathbb{R}^2$ be a set of n points. We now describe our algorithm to compute the Delaunay triangulation $\mathcal{DT}(P)$ of P. For simplicity of presentation, we assume that no four points in P are co-circular, in which case $\mathcal{DT}(P)$ is unique. The algorithm can be adopted to handle the case when four or more points are co-circular using standard techniques. We also add three points at infinity and every convex-hull vertex of P is connected to one of these points by an edge (which is a ray). These additional points and edges ensure that $\mathcal{DT}(P)$ is a triangulation of the entire plane and that each edge is adjacent to two triangles. We refer to this augmented structure as the *augmented Delaunay triangulation* of P, and we will not distiniguish between the standard and augmented Delaunay triangulation.

At a very high level, our algorithm works as follows: we randomly sample a small subset $S \subseteq P$ and compute $\mathcal{DT}(S)$ locally on one machine. For each edge $e \in \mathcal{DT}(S)$, we compute the points in P that are in *conflict* with e. We then compute the Delaunay triangulation of each such set, and discard some of the *invalid* triangles to get $\mathcal{DT}(P)$.

3.3.1 Conflict lists and kernels

Consider a subset $S \subseteq P$. Let $e = pq$ be an edge of $\mathcal{DT}(S)$, where $\triangle pqu$ and $\triangle pqv$ are the triangles adjacent to e. (Recall that each edge is adjacent to two triangles in the augmented $\mathcal{DT}(S)$.) Abusing the notation a little, by a fixed edge e we will often mean e alongwith the two triangles adjacent to e. We identify the edge pq of $\mathcal{DT}(S)$ with the four tuple $\mathsf{D}(e) = \{p, q, u, v\}$, i.e., the two endpoints of e and the two other vertices of triangles adjacent to e. The *conflict list* of e, denoted by $P_{|e}$, is the set of all those points in P that lie in the circumcircle of $\triangle pqu$ or $\triangle pqv$. The following lemma is well known (see e.g.[68]):

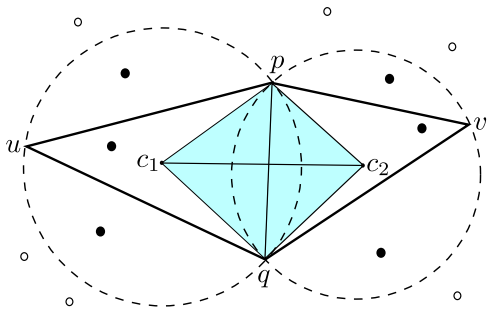


FIGURE 3.3: The kernel (in blue) for the edge pq. The points c_1 and c_2 are the circumcenters of triangles Δpqu and Δpqv resp., and c_1c_2 is the edge between $\mathrm{Vor}_S(p)$ and $\mathrm{Vor}_S(q)$. The solid points are in conflict with pq but the hollow points are not.

Lemma 18. *An edge $e = pq$ along with triangles $\triangle pqu, pqv$ appears in $\mathcal{DT}(S)$ if and only if (i) $\mathsf{D}(e) \subseteq S$ and (ii) $P_{|e} \cap S = \varnothing$.*

The *kernel* of e, $\ker(e)$, is the set of points $x \in \mathrm{Vor}_S(p) \bigcup \mathrm{Vor}_S(q)$ such that the ray $\overrightarrow{px}$ or $\overrightarrow{qx}$ intersects the edge between $\mathrm{Vor}_S(p)$ and $\mathrm{Vor}_S(q)$. See Figure 3.3.

The following lemma from [14] is the basis of our algorithm.

Lemma 19. *Let $S \subseteq P$. Then the following hold.*

(a) The set $\{\ker(e) \mid e \in \mathcal{DT}(S)\}$ partitions the plane.

(b) A triangle Δ is in $\mathcal{DT}(P)$ iff there exists an edge $e \in \mathcal{DT}(S)$ such that $\Delta \in \mathcal{DT}(P_{|e})$ and the circumcenter of Δ lies in $\ker(e)$.

Lemma 19 suggests the following algorithm: compute the Delaunay triangulation of a small sample $S \subseteq P$ locally, compute the conflict list for each edge of $\mathcal{DT}(S)$, recursively build the Delaunay triangulation for each list in parallel, and discard the triangles that do not satisfy condition (b) of Lemma 19. We formalize this algorithm later.

3.3.2 Random sampling

Consider the Delaunay triangulation of a uniformly random sample $R \subseteq P$ of size r. Consider the range space $Y = \{(P, \{P \cap \mathcal{D} \mid \mathcal{D} \text{ is an open disc}\}\}$. Y has constant VC dimension and hence R is an $O(\log(n)/r)$-net of Y with probability at least $1 - 1/n^{\Omega(1)}$ [95]. Any (open) circumcircle $\mathcal{D}$ of a triangle of $\mathcal{DT}(R)$ does not contain any points of R. Hence, by definition of an $O(\log(n)/r)$-net, we have $|P \cap \mathcal{D}| = O(n \log(n)/r)$. Thus, the size of all conflict lists is $O(n \log(n)/r)$ with probability at least $1 - 1/n^{\Omega(1)}$. To get rid of the extra $O(\log n)$ factor in the size, we do a *second level of sampling* as follows. For each edge $e \in \mathcal{DT}(R)$ with $|P_{|e}| = t_e \cdot n/r$, we sample a subset $R_{|e}$ of $O(t_e \ln t_e)$ points from $P_{|e}$ uniformly at random. We repeat both levels of sampling until $|P_{|e|e'}| = O(|P|/r)$ for every $e \in \mathcal{DT}(R), e' \in \mathcal{DT}(R_{|e})$ and $\sum_{e \in \mathcal{DT}(R)} |R_{|e}| = O(r)$. Applying Lemma 19 again gives us that a triangle $\Delta \in \mathcal{DT}(P_{|e|e'})$ is in $\mathcal{DT}(P)$ if and only if its circumcenter lies in $\ker(e) \cap \ker(e')$.

We will now show that the procedure only requires $O(\log n)$ repetitions of the two-level sampling with high probability. For an integer t, let

$$\mathcal{F}_t(R) = \{e \in \mathcal{DT}(R) \mid t \cdot n/r \leqslant |P_{|e}| < (t+1) \cdot n/r\}.$$

Lemma 20. *For all $e \in \mathcal{DT}(R)$ and all $e' \in \mathcal{DT}(R_e)$, we have $|P_{|e|e'}| = O(n/r)$ with probability $\Omega(1)$.*

Proof. For an edge $e \in \mathcal{F}_{t_e}(R)$, we sample $O(t_e \ln t_e)$ points $R_{|e}$ from $P_{|e}$. Again, $R_{|e}$ is a $O(1/t_e)$-net of $P_{|e}$ with probability $1 - 1/t_e^{\Omega(1)}$. Applying the union bound, we see that each edge in $\mathcal{DT}(R_{|e})$ has $O(|P_{|e}|/t_e) = O(n/r)$ points of $P_{|e}$ in conflict with it, with probability at least $1 - \sum_{e \in \mathcal{DT}(R)} 1/t_e^{\Omega(1)}$ which is a positive constant. □

We now show that we sample only a small number of additional points in each attempt at two-level sampling. The following exponential decay lemma, originally proved in [61, 115], is crucial for the analysis. For a parameter $r \leqslant n$, let $\phi_t(r) =$

$\mathbb{E}[|\mathcal{F}_t(R)|]$, where R is a random sample of size r. We define $\phi_{\geqslant t}(r) = \sum_{t' \geqslant t} \phi_t(r)$. We set $\phi(r) = \phi_{\geqslant 0}(r)$.

Lemma 21. *For any $t \geqslant 1$,*

$$\phi_{\geqslant t}(r) \leqslant t^{O(1)} \exp(-(t-2)) \cdot \phi(r/t).$$

Proof. The proof is adapted from [115] to our context. Let X be the set of all edges (alongwith the two adjacent triangles) that can appear in the Delaunay triangulation of a subset of P, and let $\mathsf{X}_t = \{e \in \mathsf{X} \mid |P_{|e}| \geqslant tn/r\}$.

For simplicity, we prove the lemma under a slightly different probability model, where R is not a random subset of size r but selected by choosing each point of P independently with probability $p = r/n$. The bound holds for the desired probabilistic model by a straightforward modification.

For an edge $e \in \mathsf{X}_t$, $\pi(e) = \Pr[e \in \mathcal{DT}(R)]$. By Lemma 18, $\pi(e) = p^4(1-p)^{\kappa_e}$, where $\kappa_e = |P_{|e}|$. Therefore,

$$\phi_{\geqslant t}(r) = \sum_{e \in \mathsf{X}_t} \pi(e) = \sum_{e \in \mathsf{X}_t} p^4(1-p)^{\kappa_e}.$$

We choose another random sample R' by choosing each point with probabibility $p' = p/t = \frac{r}{tn}$. Then

$$\begin{aligned}
\phi(r/t) &= \sum_{e \in \mathsf{X}} p'^4(1-p')^{\kappa_e} \\
&\geqslant \sum_{e \in \mathsf{X}_t} t^{-4} p^4 (1-p)^{\kappa_e} \left(\frac{1-p/t}{1-p}\right)^{\kappa_e} \\
&\geqslant \sum_{e \in \mathsf{X}_t} p^4 (1-p)^{\kappa_e} \psi(e),
\end{aligned}$$

where

$$\psi(e) = t^{-4} \left(\frac{1-p/t}{1-p}\right)^{\kappa_e}.$$

Assuming $p \leqslant 1/2$ and using the fact $1 - x \leqslant e^{-x}$, $1 - x \geqslant e^{-2x}$, and $\kappa_e \geqslant tn/r$ for $e \in \mathsf{X}_t$, we obtain

$$\psi(e) \geqslant t^{-4} \exp\left(\kappa_e(\frac{-2p}{t} + p)\right) \geqslant t^{-4} \exp(t-2).$$

Hence,

$$\phi_{\geqslant t}(r) \leqslant t^4 \exp(-(t-2))\phi(r/t).$$

□

We show that the Lemma 21 immediately implies that the expected additional number of points sampled is $O(r)$.

Lemma 22. *We have* $\mathbb{E}\left[\sum_{e \in \mathcal{DT}(R)} |R_{|e}|\right] = O(r)$.

Proof. We have,

$$\begin{aligned}
\mathbb{E}\left[\sum_{e \in \mathcal{DT}(R)} |R_{|e}|\right] &\leqslant \sum_{t \geqslant 0} (t+1)\ln(t+1) \cdot \phi_t(r) \\
&\leqslant \sum_{t \geqslant 0} (t+1)\ln(t+1) \cdot \phi_{\geqslant t}(r) \\
&\leqslant \phi(r) \sum_{t \geqslant 0} t^{O(1)} \ln(t+1) \exp(-t+2) \\
&\leqslant O(r).
\end{aligned}$$

□

Finally, we prove our claim.

Lemma 23. *The above procedure repeats two-level sampling at most* $O(\log n)$ *times with probability at least* $1 - 1/n^{\Omega(1)}$.

Proof. Let c a constant lower bound of the probability given in Lemma 20. By Markov's inequality, we have $\sum_{e \in \mathcal{DT}(R)} |R_{|e}| = O(r)$ with probability at least $c' > 1 - c$ by allowing the constant in the big-O to be sufficiently high. Therefore, the conditions of both Lemmas 20 and 22 are satisfied simultaneously with some constant probability. Within $O(\log n)$ repetitions of the two-level sampling, both conditions will hold with probability $1 - 1/n^{\Omega(1)}$. □

3.3.3 Algorithm

Sequential algorithm. We first briefly describe a sequential algorithm to compute the conflict lists of a sample. Given a subset $R \subseteq P$ with $|P| = n$ and $|R| = r$, we construct $\mathcal{DT}(R)$ and a point-location data structure on the triangles of $\mathcal{DT}(R)$. Using this data structure, for each $p \in P$, we determine the triangle $\Delta_p \in \mathcal{DT}(R)$ containing p. We say that a triangle Δ is in conflict with p if p lies in the interior of the circumcircle of Δ. The proof of the following well-known lemma can be found in [68].

Lemma 24. *The triangles of $\mathcal{DT}(R)$ that are in conflict with a point p form a connected subgraph in the dual graph (in the planar graph duality sense) of $\mathcal{DT}(R)$.*

The triangles in conflict with p can be found by performing a breadth-first search from Δ_p. Once these triangles are found, the edges that conflict with p can be found since they are the edges of the triangles. Point p is then added to the corresponding conflict lists.

Lemma 25. *Given a subset $R \subseteq P$ with $|P| = n$ and $|R| = r$, we can compute the conflict lists for the triangles of $\mathcal{DT}(R)$ using the above algorithm in time $O(n \log r + k)$, where k is the total size of the conflict lists.*

Proof. Computing $\mathcal{DT}(R)$ takes time $O(r \log r)$. Since there are $O(r)$ triangles in $\mathcal{DT}(R)$, the point location data structure can be constructed in $O(r \log r)$ time, and

it answers a point-location query in $O(\log r)$ time [68]. Thus, doing a point location and a subsequent breadth-first search for a point p takes time $O(\log r + k_p)$ time, where k_p is the number of edges that p is in conflict with. Summing over all points, we get the total running time to be $O(n \log r + k)$. □

Two-level sampling in the MPC model. We describe an MPC procedure called `Two-Level-Sample` (P, I), where P is a set of points distributed evenly across machines I. We need to compute the value of t_e as defined in the previous section for every edge e in $\mathcal{DT}(R)$ of a random sample $R \subseteq P$. Computing the exact values of t_e for the set $P_{|e}$ will require sending the sample R to all the machines, since P is distributed across all machines. However, the conditions of Lemmas 20 and 22 can be guaranteed even if we estimate the value of t_e to within a constant factor of the actual value.

In particular, to reduce the amount of communication, we first compute a $(1/r)$-approximation S of P using the `Sample` primitive with high probability, send it to a machine $\beta \in I$ and perform the sequential two-level sampling procedure described above on the set S. This produces $\mathcal{DT}(R)$ for a random subset $R \subseteq S$ of size r, and a second level triangulation $\mathcal{DT}(S_{|e})$ for each edge $e \in \mathcal{DT}(R)$. We send the second level triangulations to all the machines containing the set P, using the `Broadcast` primitive.

Computing the conflict lists. Each machine then does the following in parallel – for each point $p \in P$ residing on that machine, it computes the number of edges in the second level triangulations that the point p is in conflict with. Let this number be denoted by k_p. Value k_p can be computed for all p on the machine by modifying the algorithm of Lemma 25 slightly to report only the number of lists containing point p, within the same time bound. We will show later that $\sum_{\beta \in I} \sum_{p \in \beta} k_p = O(|P|)$

with high probability. Thus all the conflict lists will fit on the machines I. Let K denote an array of the values k_p, indexed by the point p. Each machine divides the portion of array K indexed by that machine's points into contiguous subarrays K_j, each summing up to cs, for some suitable constant c. For each such subarray K_j, the machine sends the points P_j corresponding to the indices of the subarray to a machine i_j. Note that $i_{j_1} \neq i_{j_2}$ for any $j_1 \neq j_2$. The machine i_j is chosen in a manner similar to the implementation of the `Partition` primitive (Section 2.2.2).

Once the points have been distributed, we compute the set of all edges in the second level triangulations that the points are in conflict with, using Lemma 25. Since for a machine i_j we have $\sum_{p \in i_j} k_p \leqslant cs$, all the edges computed for points in i_j fit on i_j. The algorithm determines if there exists a set $P_{|e|e'}$ such that $|P_{e|e'}| > c'|P|/r$ for some sufficiently large constant c'. If so, the entire algorithm restarts. We then use the `Distribute` primitive so that the set $P_{|e|e'}$ lies on a disjoint set of machines $I_{|e|e'}$. We show later that $|P_{|e|e'}| \leqslant c'|P|/r$ with high probability.

Overall MPC algorithm. We compute $\mathcal{DT}(P)$ using a recursive procedure we call `DelaunayTriangulation`, which takes as input a set of points P, machines I and a set of edges E along with their adjoining triangles, and computes $\mathcal{DT}(P)$ but retains only those triangles whose circumcenters lie in $\mathbb{R}^2 \bigcap (\bigcap_{e \in E} \ker(e))$.

If $|I| = 1$, all of the above is done sequentially. If $|I| > 1$, we first shuffle the points of P randomly across machines in I and then run `Two-Level-Sample` (P, I) to get sets $P_{|e|e'}$ on machines $I_{|e|e'}$. For each such set $P_{|e|e'}$, we recursively call `DelaunayTriangulation`$(P_{|e|e'}, I_{|e|e'}, E \cup e, e')$. The pseudocode is described in Algorithm 4.

3.3.4 Analysis

We first show that the $O(r)$ subproblems produced are of almost equal sizes.

Algorithm 4 `DelaunayTriangulation`(P, I, E)

1: If $|I| = 1$, compute $\mathcal{DT}(P)$ locally and retain those triangles whose circumcenters lie in $\cap_{e \in E} \ker(e)$.
2: Randomly shuffle the points of P across machines in I.
3: $\{(P_{|e|e'}, I_{|e|e'})\} \leftarrow$ `Two-Level-Sample` (P, I).
4: **for all** $P_{|e|e'}$ in parallel **do**
5: `DelaunayTriangulation` $(P_{|e|e'}, I_{|e|e'}, E \cup \{e, e'\})$.
6: **end for**

Lemma 26. *For any pair of edges e and e' produced in the first and second levels of sampling and triangulation on the sets S and $S_{|e}$, respectively, $|P_{|e|e'}| = O(|P|/r)$ with probability at least $1 - 1/n^{\Omega(1)}$.*

Proof. The set $P_{e|e'}$ (resp. $S_{e|e'}$) is obtained by taking all those points of P (resp. S) which lie in the conflict lists of e and e'. The region defined by these conflict lists is obtained by a finite union and intersection of disks (in particular, the union of two disks each for the conflict lists of e and e', followed by an intersection of these two regions). The VC dimension of the range space defined by such regions is a constant and so the set S is a $(1/r)$-approximation of the range space defined on the set P with probability at least $1 - 1/n^{\Omega(1)}$. Thus, since $|S_{|e|e'}|/|S| = O(1/r)$ after two-level sampling, we have that $|P_{|e|e'}|/|P| = O(1/r)$ with probability at least $1 - 1/n^{\Omega(1)}$. □

Since there are $O(r)$ subproblems, the total size of the conflict lists is $O(r) \cdot O(|P|/r) = O(|P|)$, and hence the subproblems fit on the machines I.

Lemma 27. *Procedure* `Two-level-sample` (P, I) *takes $O(1)$ rounds of computation, $O(s \log^2 n)$ time, and $O(|P| \log |P|)$ work, with probability at least $1 - 1/n^{\Omega(1)}$.*

Proof. Each of the time bounds below either hold deterministically or with probability at least $1 - 1/n^{\Omega(1)}$. Computing the set S takes $O(1)$ rounds of computation, $O(s \log s)$ time, and $O(|P| \log |P|)$ work. Two level sampling on the set S takes expected time and work $O(|S| \log r \log n) = O(s \log^2 n)$, by Lemma 25. Sending the second level of triangulations to all the machines takes one round of com-

putation, $O(s \log s)$ time, and $O(|P| \log |P|)$ work. Computing the array K takes $O(|P| \log r) = O(|P| \log |P|)$ work, by Lemma 25 and the fact that the total size of the conflict lists is $O(|P|)$. The time taken is $O(s \log r + \max_\beta \sum_{p \in \beta} k_p)$. The value of $\max_\beta \sum_{p \in \beta} k_p$ can in the worst case be $O(sr)$. However, if we randomly shuffle the points of P in the beginning across all machines, a simple Chernoff bound argument shows that $\max_\beta \sum_{p \in \beta} k_p = O(s \log |P|)$ with probability at least $1 - 1/n^{\Omega(1)}$. Goodrich et al. [91] show that random shuffling can be done in $O(1)$ rounds, $O(s \log s)$ time, and $O(|P| \log |P|)$ work. Distributing the points using the array K takes $O(1)$ rounds, $O(s \log s)$ time, and $O(|P| \log |P|)$ work. After distribution, each machine's points have conflict list size $O(s)$ and thus computing the conflict lists takes $O(s \log s)$ time and $O(|P| \log |P|)$ work, by Lemma 25. Finally, distributing the sets $P_{|e|e'}$ using the `Distribute` primitive takes $O(1)$ rounds, $O(s \log s)$ time and $O(|P| \log |P|)$ work.

Since $P_{|e|e'} = O(|P|/r)$ with probability at least $1 - 1/n^{\Omega(1)}$, the number of levels of recursion is $O(\log_r |P|) = O(1)$. By induction, each recursive level takes $O(1)$ rounds, $O(s \log^2 n)$ time, and $O(|P| \log |P|)$ work. The algorithm avoids ever restarting due to overly large conflict lists with high probability. The bounds in the statement of the lemma follow. □

The correctness of our algorithm follows from Lemma 19 and the subsections following it. We thus have the following.

Theorem 28. *Given a set P of n points in $\mathbb{R}^2$, the Delaunay triangulation of P can be built in the MPC model using $O(1)$ computation rounds, $O(s \log^2 n)$ time, and $O(n \log n)$ work, with probability at least $1 - 1/n^{\Omega(1)}$.*

3.4 Contour Tree

We now describe our algorithm to compute a contour tree of a terrain. Let $\mathbb{M} = (V, E, F)$ be a triangulation in $\mathbb{R}^2$, and let $n = |V|$. Let $h : \mathbb{M} \to \mathbb{R}$ be a piecewise linear height function over $\mathbb{M}$. We assume $s = \Omega(n^{1/2+\varepsilon})$ for some constant $\varepsilon > 0$ and that every line in $\mathbb{R}^2$ intersects $O(\sqrt{n})$ edges of $\mathbb{M}$. Let Σ be the terrain described by $\mathbb{M}$ and h.

3.4.1 Distributed contour trees

The contour tree $\mathcal{T}$ constructed by our algorithm will be stored in a distributed fashion, similar to the data structures of Chapter 2. Let $\beta \in I$ be an arbitrary machine. Given a degree-2 vertex of a tree, we say we *splice* the vertex by removing the vertex and adding an edge between its two neighbors. On β, we store a tree $\mathcal{T}_0$ formed by taking a subtree of $\mathcal{T}$ and iteratively splicing degree-2 vertices. For each vertex v of $\mathcal{T}$ that is a leaf in $\mathcal{T}_0$, let e_v be the edge incident to v in $\mathcal{T}$ separating v from other vertices in $\mathcal{T}$ that are also in $\mathcal{T}_0$. We define the *rooted subtree* of v as the connected component of $\mathcal{T} \backslash e_v$ that includes v. Let u, w be a pair of vertices in $\mathcal{T}$, and let e_u and e_w be the edges incident to u and w respectively along the unique path between u and w in $\mathcal{T}$. We define the *vine* of uw to be the connected component containing the path between u and w after removing every edge incident to u and w except e_u and e_w. Now, for each leaf v of $\mathcal{T}_0$, we recursively store the subtree $\mathcal{T}_v$ rooted at v on a subset of machines I_v (this subtree may contain only v). In addition, for each edge uw of $\mathcal{T}_0$, we recursively store the vine of uw on a subset of machines I_{uw}. See Figure 3.4. These subsets of machines are not necessarily disjoint. The subtrees are stored in such a way that for any vertex r in $\mathcal{T}_0$, number of recursively stored subtrees encountered on any r to leaf path in $\mathcal{T}$ is $O(1)$. In other words, r to leaf traversals can be performed in $O(1)$ rounds of computation and time linear in

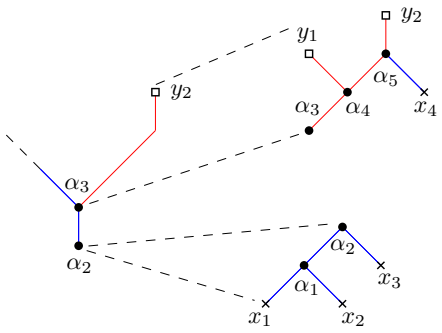


FIGURE 3.4: A possible top level subtree $\mathcal{T}_0$ for the contour tree given in Figure 3.2 is given on the left. Emanating to the right are the rooted subtree of α_2 and the vine of $\alpha_3 y_2$. The rooted subtree and vine may be stored on separate machines from $\mathcal{T}_0$.

the number of vertices along the path.

Our high-level idea for computing the tree $\mathcal{T}$ is as follows: We split $\mathbb{M}$ into pieces with approximately equal numbers of vertices. We then recursively compute distributed contour trees within each piece independently and in parallel. Merging multiple contour trees directly is difficult, so we combine the contour trees by essentially sewing their join and split trees together along piece boundaries and then running a procedure of Carr *et al.* [56]. Within a recursive computation, we *trim* excess vertices from the contour tree before sending them to their recursive parent call in order to minimize the amount of communication necessary between rounds of computation.

3.4.2 Recursive construction

We now describe our recursive algorithm in more detail. Throughout our explanation, let n be the number of triangles in the top level triangulation $\mathbb{M}$. The input to a recursive call of our algorithm is a rectangle $\square$, a triangulation $\mathbb{M}_\square = (V_\square, E_\square, F_\square)$ with one boundary component such that $\square \cap \mathbb{M}$ fits entirely within $\mathbb{M}_\square$, a piecewise linear height function $h_\square : \mathbb{M}_\square \to \mathbb{R}$, and a contiguous set of machines $I_\square = \{i_0, i_0 + 1, \dots\}$ storing $\mathbb{M}_\square$. Our algorithm is summarized as Algorithm 5. We call vertex v

of $\mathbb{M}_\square$ a boundary vertex of $\square$ if v lies on any triangle properly intersecting $\square$. Our goal is to compute the distributed contour tree $\mathcal{T}_\square$ on machines $I_\square$. We will also create tree $\mathcal{T}'$ which will be sent as the 'return value' of a recursive call. Tree $\mathcal{T}'$ contains all boundary vertices of $\mathbb{M}_\square$, and all saddle vertices v for which there exists a boundary vertex in at least three of the connected components of $\mathbb{M}_\square$ separated by the contour through v.

Algorithm 5 `ContourTree`$(\square, \mathbb{M}_\square, h_\square, I_\square)$

```
1: if |I_□| = 1 then
2:     Compute contour tree T locally.
3: else
4:     Divide □ into k = min{n^ε, |I_□|} rectangles {□_1, ..., □_k}.
5:     for all □_i in parallel do
6:         M_□i := triangles of M_□ intersecting □_i.
7:         h_□i := restriction of h_□ to M_□i.
8:         I_□i := a set of |I|/k machines.
9:         T'_□i := ContourTree(□_i, M_□i, h_□i, I_□i).
10:    end for
11:    Combine all trees T'_□i to make contour tree T.
12: end if
13: Trim T to make tree T', and return T'.
```

Tree $\mathcal{T}'$ is created from the top level structure of the distributed contour tree $\mathcal{T}_\square$ by augmenting the top level structure with any boundary vertices that are not already present, iteratively removing leaves that are not boundary vertices, and iteratively splicing degree-2 vertices that are not boundary vertices, in that order. The top level subtree given in Figure 3.4 shows the result of this procedure if α_2, α_3, and y_2 are boundary vertices. Let $\Sigma_\square$ denote the terrain described by $\mathbb{M}_\square$ and $h_\square$. The creation of $\mathcal{T}'$ mirrors the changes to $\mathcal{T}_\square$ that take place as one iteratively sets the height of all points on $\mathbb{M}_\square$ mapping to a single leaf edge to be equal to the height of the edge's non-leaf vertex. Let $\Sigma'_\square$ be the terrain created by this iterative procedure. Observe that the triangles of $\mathbb{M}_\square$ properly intersecting $\square$ have their height values unchanged between $\Sigma_\square$ and $\Sigma'_\square$. The portions of $\mathcal{T}_\square$ removed from $\mathcal{T}'$ can be represented as rooted subtrees and vines. The top level structures for these rooted subtrees and

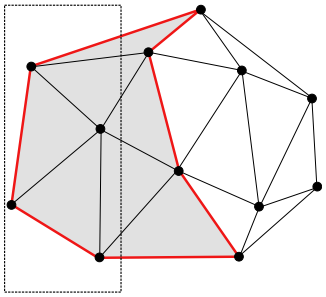


FIGURE 3.5: One rectangle $\square_i$ and its associated triangles $\mathbb{M}_i$.

vines are permanently stored on machines in $I_\square$. Note that $\mathcal{T}'$ is still augmented with every boundary vertex, although these boundary vertices may be degree-2.

Lemma 29. *The number of vertices in $\mathcal{T}'$ is $O(\sqrt{n})$.*

Proof. By our assumption that each line in $\mathbb{R}^2$ crosses $O(\sqrt{n})$ edges, the boundary of $\mathbb{M}_\square$ crosses $O(\sqrt{n})$ triangles and there are therefore $O(\sqrt{n})$ boundary vertices. Every degree-1 or degree-2 vertex in $\mathcal{T}'$ is a boundary vertex. The lemma follows. □

We now describe how to compute the distributed contour tree $\mathcal{T}_\square$. If $|I_\square| = 1$, then we compute $\mathcal{T}_\square$ sequentially using the algorithm of Carr *et al.* [56]. Otherwise, we do so recursively. Let n' be the number of vertices of $\mathbb{M}$ strictly interior to $\square$. We begin by dividing $\square$ along vertical lines into $k = \min\{n^\varepsilon, |I_\square|\}$ rectangles $\{\square_1, \ldots, \square_k\}$. These lines are chosen so that there exist at most $\lceil n'/k \rceil$ vertices of $\mathbb{M}$ in each rectangle $\square_i$. Finding the lines can be done in $O(1)$ rounds, $O(s \log s)$ time, and $O(n' \log n')$ work by sorting the vertices inside $\square$ by their x-coordinate using a procedure of Goodrich [89] and looking at the $i \cdot \lceil n'/k \rceil$-th vertex for each i in this order. Now, let Π be a function from $F_\square$ to subsets of $\{1, \ldots, k\}$ where for each triangle Δ of $\mathbb{M}_\square$, $\Pi(\Delta)$ is the subset of rectangles $\square_i$ that intersect Δ. We run `Distribute`$(F_\square, I_\square, \Pi)$ to distribute copies of the triangles to disjoint sets of machines.

Let $I_{\square_i}$ for $i \in \{1, \ldots, k\}$ denote the disjoint subsets of machines that receive

triangles from the `Distribute` primitive. Let $\mathbb{M}_{\square_i}$ be the set of triangles sent to machines $I_{\square_i}$. See Figure 3.5. We recursively compute distributed contour trees of each triangulation $\mathbb{M}_{\square_i}$ using the natural restrictions of h to those triangulations. In the following lemma, we prove that the distribution step above is doable in the MPC model. In the sequel, we describe how to combine the trees returned by each recursive call to create the top level subtree for $\mathcal{T}_{\square}$.

Lemma 30. *Our algorithm meets the preconditions of the* `Distribute` *primitive.*

Proof. By assumption, $I_{\square}$ is a contiguous set of machines as required. We meet the requirement that $k = O(s^{1/2})$ as $s = \Omega(n^{1/2+\varepsilon})$. Similarly, we meet the requirement that $k \leqslant |I|$. Each rectangle $\square_i$ contains $O(n'/k)$ points, so it receives $O(n'/k)$ *non*-boundary triangles. Further, by the assumption that each line in $\mathbb{R}^2$ intersects at most $O(\sqrt{n})$ edges, there are at most $O(\sqrt{n}) = O(n'/k)$ boundary triangles for each $\square_i$. Together, these facts imply that for each $i \in \{1, \ldots, k\}$, there are at most $O(n'/k)$ triangles Δ for which $\Pi(\Delta) = i$ as required. Finally, each triangle can only be copied at most $n^{\varepsilon} = O(s)$ times as required. □

3.4.3 Combining multiple contour trees

Let $\mathcal{T}_{\square_i}$ be the contour tree for $\mathbb{M}_{\square_i}$, and let $\mathcal{T}'_{\square_i}$ be the tree returned by the recursive procedure computing $\mathcal{T}_{\square_i}$. We send all of these trees to a single machine $\beta \in I_{\square}$ in $O(1)$ rounds of computation. By Lemma 29, each such tree has size $O(\sqrt{n})$, so they all fit on machine β simultaneously. The rest of the computation occurs sequentially on machine β.

Recall, $\mathcal{T}'_{\square_i}$ is the contour tree for a surface $\Sigma'_{\square_i}$. We create the top level structure for $\mathcal{T}_{\square}$ using the following procedure: Let $\Sigma''_{\square}$ be the terrain that results from replacing the portion of $\Sigma_{\square}$ within each rectangle $\square_i$ with the portion of $\Sigma'_{\square_i}$ inside $\square_i$. We say vertex v is *important* if it appears as a vertex of any tree $\mathcal{T}'_{\square_i}$.

We use the following procedure to compute the join tree $\mathcal{J}''_{\square}$ of $\Sigma''_{\square}$. We compute the join tree $\mathcal{J}'_{\square_i}$ for each contour tree $\mathcal{T}'_{\square_i}$. Next, we sort the entire collection of important vertices in increasing order of height. We then use a set union-find data structure [64] to track the sublevel sets as we consider vertices of increasing height values. The sets of the union-find data structure will be collections of important vertices, and a find operation on an important vertex v will return the highest important vertex v' in the same sublevel set component as v. For each important vertex v in increasing order by height, we look at its down neighbors from each of the join trees $\mathcal{J}'_{\square_i}$ that contain v and do a find operation. The down neighbors of v in $\mathcal{J}''_{\square}$ will be all the vertices returned by the find operations.

Lemma 31. *The previous algorithm correctly computes the join tree $\mathcal{J}''_{\square}$ of $\Sigma''_{\square}$.*

Proof. We prove the lemma by induction on the increasing heights of important vertices. Let v be an important vertex, and consider any sublevel set component C in $\Sigma''_{\square}$ of height $h(v)-\delta$ for an arbitrarily small δ such that C lies within the sublevel set component of v (assuming C exists). Vertex v contains a neighbor u that lies in C. Let $\mathbb{M}_{\square_i}$ be a set of triangles containing edge (v,u). Let v' be the highest important vertex of $\Sigma'_{\square_i}$ with height strictly below v that shares a sublevel component of $\Sigma'_{\square_i}$ with v and u. Vertex v' is a down neighbor of v in $\mathcal{J}'_{\square_i}$ that lies in component C. By induction, the find operation on v' will return the highest important vertex of C, which the algorithm will correctly assign as a down neighbor of v in $\mathcal{J}''_{\square}$. □

The split tree $\mathcal{S}''_{\square}$ of $\Sigma''_{\square}$ can be computed using a similar procedure. Finally, we use the procedure of Carr *et al.* [56] to create the contour tree $\mathcal{T}''_{\square}$ of $\Sigma''_{\square}$. Tree $\mathcal{T}''_{\square}$ is the top level structure used to store $\mathcal{T}_{\square}$ in a distributed manner.

Lemma 32. *Procedure* `ContourTree` *takes $O(1)$ rounds of computation, $O(s \log s)$ time, and $O(n \log n)$ work.*

Proof. The number of levels of recursion of the procedure is $O(\log_{n^\varepsilon} n) = O(1)$, and each level of recursion uses only $O(1)$ rounds of computation. For time and work, the most expensive operation is building the join and split trees to compute a contour tree $\mathcal{T}''_\square$. These operations require sorting $O(s)$ vertices in $O(s \log s)$ time. Summing over all the vertices stored on all machines, the total amount of work is $O(n \log n)$. □

We thus have the following.

Theorem 33. *Let $\mathbb{M} = (V, E, F)$ be a triangulation in $\mathbb{R}^2$ such that every line in $\mathbb{R}^2$ intersects $O(\sqrt{n})$ edges of $\mathbb{M}$ where $n = |V|$. Let $h : \mathbb{M} \to \mathbb{R}$ be a piecewise linear height function on $\mathbb{M}$. The distributed contour tree $\mathcal{T}$ of h can be built in the MPC model in $O(1)$ computation rounds, $O(s \log s)$ time, and $O(n \log n)$ work, assuming $s = \Omega(n^{1/2+\varepsilon})$ for some constant $\varepsilon > 0$.*

3.5 Conclusion

We described two algorithms for the MPC model that aid in terrain analysis. The first was a Las Vegas algorithm to compute a Delaunay triangulation of an arbitrary point set in $\mathbb{R}^2$. With high probability, it runs in $O(1)$ rounds of computation, and it uses only $O(s \log^2 n)$ time and $O(n \log n)$ work. The second algorithm deterministically constructs a contour tree of a triangulation $\mathbb{M}$, assuming the machines have sufficient memory and $\mathbb{M}$ is well-formed. This algorithm required only $O(1)$ rounds of computation, $O(s \log s)$ time, and $O(n \log n)$ work. We leave the removal of these restrictions from our contour tree construction algorithm as an interesting open problem. For an arbitrary triangulation $\mathbb{M}$, a possible approach is to build a small, well-balanced planar separator with small boundary, build the contour tree for each piece in parallel, and sew them along the boundary. However, computing a separator in the MPC model appears difficult, and will require new ideas.

4

Comparing Topological Descriptors and Metric Spaces

4.1 Introduction

The Gromov-Hausdorff distance (or GH distance for brevity) [92] is one of the most natural distance measures between metric spaces, and has been used, for example, for matching deformable shapes [119, 48], and for analyzing hierarchical clustering trees [55]. Informally, the Gromov-Hausdorff distance measures the *additive* distortion suffered when mapping one metric space to another using a correspondence between their points. Even when working with spaces of equal size, correspondences such as those computed for GH distance are often preferable to, say, bijections, because the distance is less sensitive to slight stretching or the exact choice of points sampled from a larger space (see Figure 4.1). Multiple approaches have been proposed to estimate the Gromov-Hausdorff distance [119, 48, 118].

Despite much effort, the problem of computing, either exactly or approximately, GH distance has remained elusive. The problem is not known to be NP-hard, and

computing the GH distance, even approximately, for *graphic metrics*[1] is at least as hard as the graph isomorphism problem. Indeed, the metrics for two graphs have GH distance 0 if and only if the two graphs are isomorphic. Motivated by this trivial hardness result, it is natural to ask whether GH distance becomes easier in more restrictive settings such as geodesic metrics over trees, where efficient algorithms are known for checking isomorphism [24].

4.1.1 Related work

Most prior work has either focused on embedding one metric space into another, or computing a bijection between two equal-sized finite metric spaces that minimizes the multiplicative distortion (see Section 1.3).

The work closest to computing the GH distance is the *additive distortion problem* [94] – given two equal-sized point sets $S, T \subset \mathbb{R}^d$, find the smallest Δ such that there exists a bijection $f : S \to T$ such that $|d(x, y) - d(f(x), f(y))| \leqslant \Delta$. They show that it is NP-hard to approximate by a factor better than 3 in $\mathbb{R}^3$, and also give a 2-approximation for $\mathbb{R}^1$ and a 5-approximation for the more general problem of embedding an arbitrary metric space onto $\mathbb{R}^1$. However, their setting differs from ours in two major ways – firstly, they consider finite metric spaces of equal size, whereas in our work the metric spaces may be uncountably infinite; secondly, they consider bijections between metric spaces, whereas in our work we deal with correspondences between metric spaces which are more general than bijections. In particular, an optimal correspondence between arbitrary spaces of equal size may not be a bijection (see Figure 4.1), so these hardness results do not apply to GH distance. Thus, their approach cannot be easily extended to our setting.

The interleaving distance between merge trees [121] was proposed as a measure

[1] A graphic metric measures the shortest path distance between vertices of a graph with unit length edges.

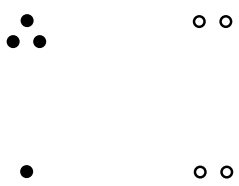


FIGURE 4.1: Solid and hollow points sampled from distinct metric spaces; each metric space contains two distinct clusters with similar intra- and inter-cluster distances, but the points are sampled differently for each. Using correspondences between the two sets will lead to a small distortion. However, using bijections will map points from the same cluster to different clusters in the other metric space, leading to a much larger distortion.

to compare functions over topological domains that is stable to small perturbations in a function. Distances for the more general Reeb graphs are given in [38, 69]. These concepts are related to the GH distance (Section 4.4), which we will leverage to design an approximation algorithm for the GH distance for metric trees.

4.1.2 Our results

In this chapter, we give the first non-trivial results on approximating the GH distance between metric trees. First, we prove (in Section 4.3) that the problem remains NP-hard even for metric trees via a reduction from 3-PARTITION. In fact, we show that there exists no efficient algorithm with approximation ratio less than 3 unless P = NP. As noted above, we are not aware of any result that shows the GH distance problem being NP-hard even for general graphic metrics.

To complement our hardness result, we give an $O(\sqrt{n})$-approximation algorithm for the GH distance between metric trees with n nodes and *unit length* edges. Our algorithm works with arbitrary edge lengths as well; however, the approximation ratio becomes $O(\min\{n, \sqrt{rn}\})$ where r is the ratio of the longest edge length in both trees to the shortest edge length. Even achieving the $O(n)$-approximation ratio presented here for arbitrary r is a non-trivial task, and we hope this first non-trivial approximation result will stimulate further research on this and similar problems.

Our algorithm uses a reduction, described in Section 4.4, to the similar problem of computing the *interleaving distance* [121] between two *merge trees*. Given a function $f : \mathbb{X} \to \mathbb{R}$ over a topological space $\mathbb{X}$, the merge tree T_f describes the connectivity between components of the sublevel sets of f (see Section 4.2 for a more formal definition). Morozov et al. [121] proposed the interleaving distance as a way to compare merge trees and their associated functions[2]. For us, interleaving distance provides a convenient way to measure distance after *rooting* the input trees so that correspondences are more nicely structured. We describe, in Section 4.5, an $O(\min\{n, \sqrt{rn}\})$-approximation algorithm for interleaving distance between merge trees, and our reduction provides a similar approximation for computing the GH distance between two metric trees.

Note on independent and follow-up work. After writing preliminary versions of the current work, we became aware of recent work by Schmiedl [135] in which he proves a lower bound of 3 for the approximation ratio of any polynomial time approximation for GH distance in metric trees. As in our proof, his proof ultimately comes down to a reduction from 3-Partition. He also gives an approximation algorithm for arbitrary metrics, but the running time is not guaranteed to be polynomial in the size of the input.

There is also follow-up work [140] on a fixed-parameter tractable algorithm for approximating the GH distance between tree metrics to a constant factor.

4.2 Preliminaries

Metric spaces and the Gromov-Hausdorff distance. A *metric space* $\mathcal{X} = (X, \rho)$ consists of a (potentially infinite) set X and a function $\rho : X \times X \to \mathbb{R}_{\geqslant 0}$ such that the

[2] In fact, our hardness result can be easily extended to the GH distance between graphic metrics for trees and the interleaving distance between merge trees.

following hold: $\rho(x, y) = 0$ iff $x = y$; $\rho(x, y) = \rho(y, x)$; and $\rho(x, z) \leqslant \rho(x, y) + \rho(y, z)$.

Given sets A and B, a *correspondence* between A and B is a set $\mathcal{C} \subseteq A \times B$ such that: (i) for all $a \in A$, there exists $b \in B$ such that $(a, b) \in \mathcal{C}$; and (ii) for all $b \in B$, there exists $a \in A$ such that $(a, b) \in \mathcal{C}$. We use $\Pi(A, B)$ to denote the set of all correspondences between A and B.

Let $\mathcal{X}_1 = (X_1, \rho_1)$ and $\mathcal{X}_2 = (X_2, \rho_2)$ be two metric spaces. The *distortion* of a correspondence $\mathcal{C} \in \Pi(X_1, X_2)$ is defined as:

$$\text{Dist}(\mathcal{C}) = \sup_{(x,y),(x',y') \in \mathcal{C}} |\rho_1(x, x') - \rho_2(y, y')|.$$

The *Gromov-Hausdorff distance* [118], d_{GH}, between $\mathcal{X}_1$ and $\mathcal{X}_2$ is defined as:

$$d_{GH}(\mathcal{X}_1, \mathcal{X}_2) = \frac{1}{2} \inf_{\mathcal{C} \in \Pi(X_1, X_2)} \text{Dist}(\mathcal{C}).$$

Intuitively, d_{GH} measures how close can we can get to an *isometric* (distance-preserving) embedding between two metric spaces. We note that there are different equivalent definitions of the Gromov-Hausdorff distance; see e.g, Theorem 7.3.25 of [54] and Remark 1 of [118].

A *tree* $T = (V, E)$ consists of a set of *nodes* V and *edges* E connecting pairs of nodes such that some geometric realization of T is simply connected. Note our definition allows for nodes of degree 2. The *leaves* of T are the nodes of degree 1, and its *branching nodes* are the nodes of degree 3 or higher.

Given a tree $T = (V, E)$ and a length function $l : E \to \mathbb{R}_{\geqslant 0}$, we associate a metric space $\mathcal{T} = (\mathbf{T}, d)$ with T as follows. $\mathbf{T}$ is a geometric realization of T. For $x, y \in \mathbf{T}$, define $d(x, y)$ to be the length of the unique shortest path $\pi(x, y) \in \mathbf{T}$. It is clear that d is a metric. The metric space thus obtained is a *metric tree*. We often do not distinguish between T and $\mathbf{T}$ and write $\mathcal{T} = (T, d)$.

Merge trees and the interleaving distance. Let $f : \mathbb{X} \to \mathbb{R}$ be a continuous function from a connected topological space $\mathbb{X}$ to the set of real numbers. The *sublevel set*

at a value $a \in \mathbb{R}$ is defined as $f_{\leqslant a} = \{x \in \mathbb{X} \mid f(x) \leqslant a\}$. A *merge tree* M_f captures the evolution of the topology of the sublevel sets as the function value is increased continuously from $-\infty$ to $+\infty$. Formally, it is obtained as follows. Let epi $f = \{(x, y) \in \mathbb{X} \times \mathbb{R} \mid y \geqslant f(x)\}$. Let $\bar{f} : \text{epi } f \to \mathbb{R}$ be such that $\bar{f}((x, y)) = y$. We may say $\bar{f}((x, y))$ is the *height* of point $(x, y) \in \mathbb{X} \times \mathbb{R}$. For two points (x, y) and (x', y') in $\mathbb{X} \times \mathbb{R}$ with $y = y'$, let $(x, y) \sim (x', y')$ denote them lying in the same component of $\bar{f}^{-1}(y)(= \bar{f}^{-1}(y'))$. Then $\sim$ is an equivalence relation, and the merge tree M_f is defined as the quotient space $(\mathbb{X} \times \mathbb{R})/\sim$.

The sublevel sets f_a of f merge as function value a increases, so we get a rooted tree where branching nodes represent the moments where two components merge and leaves represent the birth of a new component at a local minimum. Figure 4.2 shows an example of a merge tree for a 1-dimensional function. Note that the merge tree extends to a height of ∞. Our assumption that $\mathbb{X}$ is connected implies we have only one component in $f_{\leqslant N}$ for some sufficiently large $N > 0$, and M_f is, in fact, a single tree.

We define the *root* of merge tree M_f to be the node with the highest function value. The *ancestors* of a point $x \in \mathsf{M}_f$ are the points lying on the unique path from x to the root of M_f as well as all points with greater height than the root. We define the *length* of any edge in a merge tree (other than the edge to infinity) to be the height difference between its two endpoints. Ancestors of points in M_g are defined similarly.

Since each point $x \in \mathsf{M}_f$ represents a component of a sublevel set at a certain height, we can associate this height value with x, denoted by $\hat{f}(x)$. Given a merge tree M_f and $\varepsilon \geqslant 0$, an ε-shift map $\sigma_f^\varepsilon : \mathsf{M}_f \to \mathsf{M}_f$ is the map that maps a point $x \in \mathsf{M}_f$ to its ancestor at height $\hat{f}(x) + \varepsilon$, i.e., $\hat{f}(\sigma_f^\varepsilon(x)) = \hat{f}(x) + \varepsilon$. Given $\varepsilon \geqslant 0$ and merge trees M_f and M_g, two continuous maps $\alpha : \mathsf{M}_f \to \mathsf{M}_g$ and $\beta : \mathsf{M}_g \to \mathsf{M}_f$ are

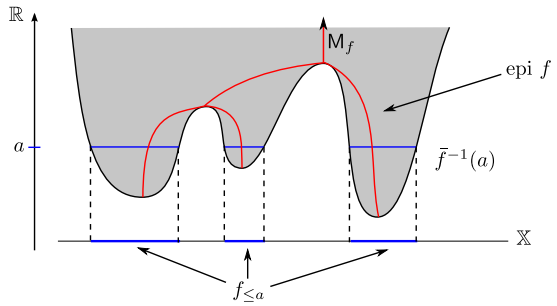


FIGURE 4.2: Merge tree M_f (shown in red) for a function $f : \mathbb{X} \to \mathbb{R}$, where $\mathbb{X} = \mathbb{R}$. epi f is shown in grey, and $\bar{f}^{-1}(a)$ is the grey region below the blue horizontal lines in epi f.

said to be ε-compatible if they satisfy the following conditions :

$$\begin{aligned} &\hat{g}(\alpha(x)) = \hat{f}(x) + \varepsilon, \forall x \in \mathsf{M}_f; \quad \hat{f}(\beta(y)) = \hat{g}(y) + \varepsilon, \forall y \in \mathsf{M}_g; \\ &\beta \circ \alpha = \sigma_f^{2\varepsilon}; \qquad\qquad \alpha \circ \beta = \sigma_g^{2\varepsilon}. \end{aligned} \tag{4.1}$$

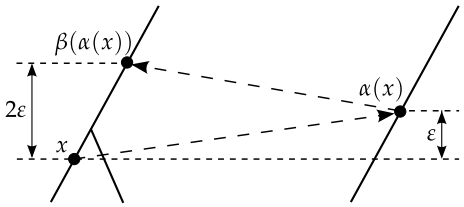


FIGURE 4.3: Part of trees M_f and M_g showing α and β.

See Figure 4.3 for an example. The *interleaving distance* [121] is then defined as

$$d_I(\mathsf{M}_f, \mathsf{M}_g) = \inf\{\varepsilon \geqslant 0 \mid \text{there exist } \varepsilon\text{-compatible maps } \alpha \text{ and } \beta\}.$$

Remark. We can relax the requirements on α and β from their normal definitions as follows.

(i) Instead of requiring *exact* value changes, we require

$$\hat{f}(x) \leqslant \hat{g}(\alpha(x)) \leqslant \hat{f}(x) + \varepsilon, \forall x \in \mathsf{M}_f; \quad \hat{g}(y) \leqslant \hat{f}(\beta(y)) \leqslant \hat{g}(y) + \varepsilon, \forall y \in \mathsf{M}_g.$$

(ii) If x_1 is an ancestor of x_2 in M_f, then $\alpha(x_1)$ is an ancestor of $\alpha(x_2)$ in M_g. A similar rule applies for β.

(iii) $\beta(\alpha(x))$ must must be mapped to an ancestor of x and $\alpha(\beta(y))$ must be mapped to an ancestor of y.

Any pair of maps satisfying the original requirements also satisfies the relaxed requirements for the same value of ε. Conversely, for any pair of maps satisfying the relaxed requirements, we can *stretch up* the images for each map by instead mapping to the ancestors of the appropriate height of the original images, without changing the value of ε. Thus, both definitions of interleaving distance are equivalent. For convenience, when two ε-compatible maps are given to us we assume that they satisfy (4.1), but we construct ε-compatible maps that satisfy the relaxed conditions mentioned, knowing that they can be "stretched" as just described to satisfy (4.1).

If we know $\alpha(x)$ for a point x at height h, then we can compute $\alpha(y)$ for any ancestor y of x at height $h' \geqslant h$ by simply putting $\alpha(y) = \sigma_f^{h'-h}(\alpha(x))$. A similar claim holds for β. Thus specifying the maps for the leaves of the trees suffices, because any point in the tree is the ancestor of at least one of the leaves. Hence, these maps have a representation that requires linear space in the size of the trees.

Morozov *et al.* [121] prove several facts about interleaving distance that will prove useful in this work. In particular, interleaving distance is a metric [121, Lemma 1]. Further, it is a *stable* distance measure with respect to small function perturbations; specifically, given two functions $f : \mathbb{X} \to \mathbb{R}$ and $g : \mathbb{X} \to \mathbb{R}$, we have $d_I(\mathsf{M}_f, \mathsf{M}_g) \leqslant \|f - g\|_\infty$ [121, Theorem 2].

4.3 Hardness of Approximation

We now show the hardness of approximating the GH distance by a reduction from the following decision problem called *balanced partition* (or BAL-PART for brevity): given a multiset of positive integers $X = \{a_1, \ldots, a_n\}$, and an integer m such that $1 \leqslant m \leqslant n$, is it possible to partition X into m multisets $\{X_1, \ldots, X_m\}$ such that all

the elements in each multiset sum to the same quantity $\mu = \left(\sum_{i=1}^{n} a_i\right)/m$? We prove below that BAL-PART is *strongly* NP-complete, i.e., it remains NP-complete even if $a_i \leqslant n^c$ for some constant $c \geqslant 1$ for all $1 \leqslant i \leqslant n$.

Lemma 34. BAL-PART *is strongly NP-complete.*

Proof. We reduce 3-PARTITION, a strongly NP-complete problem [86] to BAL-PART. Given a multiset of positive integers $Y = \{a_1, \ldots, a_n\}$ with $n = 3m$, 3-PARTITION asks to partition Y into m multisets $\{Y_1, \ldots, Y_m\}$ of size 3 each so that the elements in each multiset sum to the same quantity. Given a 3-PARTITION instance, we construct an instance of BAL-PART as follows.

Basically, we add a sufficiently large number to each a_i so that if two multisets of the new numbers have the same sum, they have the same number of elements. In particular, let $\bar{a} = \sum_{i=1}^{n} a_i$. Then set $a'_i = a_i + \bar{a}$ and $X = \{a'_1, \ldots, a'_n\}$. This reduction takes polynomial time, and the new numbers are polynomially larger than the original ones. We show that there exists an appropriate partition of Y iff there exists an appropriate partition of X.

Suppose there exists an appropriate partition $\{Y_1, \ldots, Y_m\}$ of Y. Then setting $X_i = \{a'_j \mid a_j \in Y_i\}$ for $i = 1, \ldots, m$ gives us the desired partition of X.

Conversely, suppose there exists an appropriate partition $\{X_1, \ldots, X_m\}$ of X. Suppose $|X_i| = n_1 > |X_j| = n_2$ for some $i \neq j$. We thus have

$$\sum_{a'_k \in X_i} a_k + n_1\bar{a} = \sum_{a'_k \in X_j} a_k + n_2\bar{a} \Rightarrow (n_1 - n_2)\bar{a} = \sum_{a'_k \in X_j} a_k - \sum_{a'_k \in X_i} a_k \Rightarrow \sum_{a'_k \in X_j} a_k - \sum_{a'_k \in X_i} a_k \geqslant \bar{a}, \tag{4.2}$$

a contradiction since $\sum_{a'_k \in X_j} a_k < \bar{a}$. Thus, each partition X_i is of equal size. Since $n = 3m$, the size of each X_i is 3. □

We now reduce an instance of BAL-PART, in which each $a_i \leqslant n^c$ for some constant $c \geqslant 1$, to GH-distance computation. Consider an instance $X = \{a_1, \ldots, a_n\}$

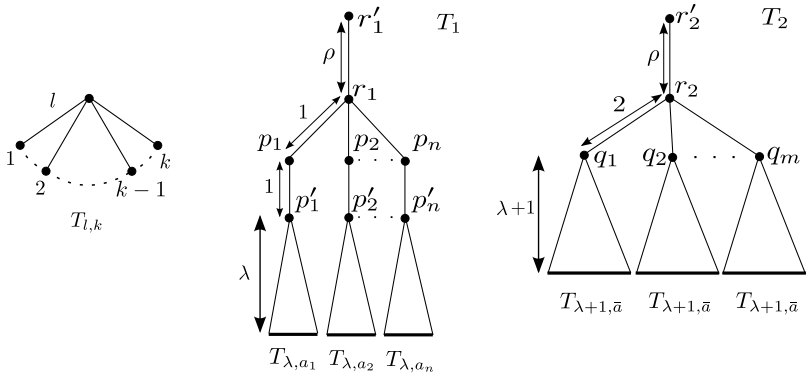


FIGURE 4.4: The trees $T_{l,k}$, T_1 and T_2.

and $1 \leqslant m \leqslant n$ of BAL-PART. If $m = 1$, then a balanced partition into one multiset trivially exists. If $m > 1$, we construct two trees T_1 and T_2 as follows. Let $\lambda > 8$ and $\rho < \lambda - 6$ be two positive constants. Let $T_{l,k}$ denote a star graph having k edges, each of length l. T_1 has a node r_1 incident to an edge (r_1, r'_1) of length ρ and to n edges $\{(r_1, p_1), \ldots, (r_1, p_n)\}$ of length 1. Each node p_i is incident to another edge (p_i, p'_i) of length 1, and each p'_i is the center of a copy of T_{λ,a_i}. T_2 consists of a node r_2 incident to an edge (r_2, r'_2) of length ρ and to m edges $\{(r_2, q_1), \ldots, (r_2, q_m)\}$ of length 2, where each q_i is the center of a distinct copy of $T_{\lambda+1,\bar{a}}$, and $\bar{a} = \left(\sum_{i=1}^{n} a_i\right)/m$. See Figure 4.4 for an illustration. We refer to the edges of each T_{λ,a_i} in T_1 and the edges of copies of $T_{\lambda+1,\bar{a}}$ in T_2 as *bottom* edges. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ denote the metric trees associated with T_1 and T_2 respectively. Since λ, ρ are constants and $a_i \leqslant n^c$ for all $1 \leqslant i \leqslant n$, this construction can be done in polynomial time.

Lemma 35. *If (X, m) is a* yes *instance of* BAL-PART*, then $d_{GH}(\mathcal{T}_1, \mathcal{T}_2) \leqslant 1$. Otherwise, $d_{GH}(\mathcal{T}_1, \mathcal{T}_2) \geqslant 3$.*

Proof. Suppose X can be partitioned into m subsets $X_1, \ldots, X_m$ of equal weight $\bar{a} = \left(\sum_{i=1}^{n} a_i\right)/m$. We construct a correspondence $\mathcal{C}$ between $\mathcal{T}_1$ and $\mathcal{T}_2$ with distortion at most 2, implying that $d_{GH}(\mathcal{T}_1, \mathcal{T}_2) \leqslant 1$. A linearly interpolated bijection between the points of edges (r_1, r'_1) and (r_2, r'_2), with r_1 mapping to r_2 and r'_1 mapping to r'_2,

is added to $\mathcal{C}$. If $a_i \in X_j$, the linearly interpolated bijection between edges (r_1, p_i) and (r_2, q_j) is added to $\mathcal{C}$, with p_i mapping to q_j. Also, using linear interpolation, we map the *path* from p_i to each leaf of T_{λ,a_i} to distinct *edges* going from q_j to leaves of its copy of $T_{\lambda+1,\bar{a}}$; this is possible since $T_{\lambda+1,\bar{a}}$ has $\bar{a}$ leaves, and $\sum_{a\in X_j} a = \bar{a}$. Note that each edge (p_i, p_i') may be mapped to a contiguous part of multiple edges in $\mathcal{T}_2$. Overall, the distortion induced by $\mathcal{C}$ is at most 2; this stems from the fact that $\mathcal{C}$ is piecewise linear, and the difference between the length of any node-to-node path in one tree and its image under $\mathcal{C}$ in the other tree is at most 2.

Suppose $d_{GH}(\mathcal{T}_1, \mathcal{T}_2) < 3$, and let $\mathcal{C}$ be a correspondence between $\mathcal{T}_1$ and $\mathcal{T}_2$ with distortion < 6. Consider two leaves $l_1, l_2 \neq r_2'$ in $\mathcal{T}_2$. Then $d(l_1, l_2) \geqslant 2\lambda + 2$. Let l_1', l_2' be their corresponding images in $\mathcal{T}_1$ under $\mathcal{C}$. We argue that l_1', l_2' lie on distinct bottom edges of $\mathcal{T}_1$. Indeed, since $\mathrm{Dist}(\mathcal{C}) < 6$, the distance between l_1' and l_2' is $d(l_1', l_2') > d(l_1, l_2) - 6 \geqslant 2\lambda - 4$. If l_1', l_2' lie on the same edge of $\mathcal{T}_1$, then $d(l_1', l_2') \leqslant \lambda < 2\lambda - 4$, so they have to lie on distinct edges of $\mathcal{T}_1$. If either of l_1', l_2' lies on an edge r_1p_i, for some $i \leqslant n$, then by construction, $d(l_1', l_2') \leqslant \lambda + 4 < 2\lambda - 4$ (recall that $\lambda > 8$). Finally, if either of l_1', l_2' lies on (r_1, r_1') then by the choice of ρ, $d(l_1', l_2') \leqslant \rho + \lambda + 2 < 2\lambda - 4$. Thus, both l_1' and l_2' lie on distinct bottom edges of $\mathcal{T}_1$. In particular, the corresponding image l_1' in $\mathcal{T}_1$ of any *one* leaf l_1 in $\mathcal{T}_2$ must lie on a bottom edge of $\mathcal{T}_1$, because there exists some other leaf l_2 in $\mathcal{T}_2$ by the assumption that $m > 1$. Hence, $\mathcal{C}$ induces a bijection χ between the leaves of $\mathcal{T}_1$ and $\mathcal{T}_2$, where $\chi(l) = l'$ for $l \in \mathcal{T}_2$ and $l' \in \mathcal{T}_1$ is the leaf whose incident edge contains the image(s) of l under $\mathcal{C}$. Note that if $l_i, l_j \in \mathcal{T}_2$ are incident to q_i, q_j with $q_i \neq q_j$, then $\chi(l_i)$ and $\chi(l_j)$ are incident to $p_{i'}, p_{j'}$ with $p_{i'} \neq p_{j'}$; otherwise $d(l_i, l_j) = 2\lambda + 6$ and $d(\chi(l_i), \chi(l_j)) \leqslant 2\lambda$, thereby incurring a distortion of at least 6. Hence, the bijection χ can be used to partition X into m subsets $X_1, \ldots, X_m$ of equal weight as follows : if $\chi(l) = l'$ for l incident to q_i and l' incident to p_j, then $a_j \in X_i$. Thus, (X, m) is a

yes instance of BAL-PART. □

We may also apply the reduction to metric trees with unit edge lengths by subdividing longer edges with an appropriate number of vertices. We thus have the following theorem.

Theorem 36. *Unless* P = NP*, there is no polynomial-time algorithm to approximate the Gromov-Hausdorff distance between two metric trees to a factor better than 3, even in the case of metric trees with unit edge lengths.*

4.4 Gromov-Hausdorff and Interleaving Distances

In this section we show that the GH distance between two tree metric spaces $\mathcal{T}_1$ and $\mathcal{T}_2$, and the interleaving distance between two appropriately defined trees induced by the $\mathcal{T}_i$'s are within constant factors of each other.

Given a metric tree $\mathcal{T} = (T, d)$, let $V(T)$ denote the nodes of the tree. Given a point $s \in T$ (not necessarily a node), let $f_s : T \to \mathbb{R}$ be defined as $f_s(x) = -\, d(s, x)$. Equipped with this function, we obtain a merge tree T^s from $\mathcal{T}$. Intuitively, T^s has the structure of rooting T at s, and then adding an extra edge incident to s with function value extending from 0 to $+\infty$. If s is an internal node of T or an interior point of an edge of T, s remains the root of T^s. But if s is a leaf of T, then s gets merged with the infinite edge and the node of T adjacent to s becomes the root of T^s.

Let $\mathcal{T}_1 = (T_1, d_1)$ and $\mathcal{T}_2 = (T_2, d_2)$ be two metric trees. Define

$$\Delta = \min_{u \in V(T_1), v \in V(T_2)} d_I(T_1^u, T_2^v). \tag{4.3}$$

We prove that Δ is within a constant factor of $d_{GH}(\mathcal{T}_1, \mathcal{T}_2)$. We first prove a lower bound on Δ.

Lemma 37. $\frac{1}{2} d_{GH}(\mathcal{T}_1, \mathcal{T}_2) \leqslant \Delta$.

Proof. Suppose $\Delta = d_I(T_1^s, T_2^t)$ for some $s \in V(T_1)$ and $t \in V(T_2)$. Set $f := f_s$ and $g := f_t$. Let $\alpha : T_1^s \to T_2^t, \beta : T_2^t \to T_1^s$ be Δ-compatible maps. We define the functions $\alpha^* : T_1^s \to T_2^t$ and $\beta^* : T_2^t \to T_1^s$ as follows :

$$\alpha^*(x) = \begin{cases} \alpha(x) & \text{if } g(\alpha(x)) \leqslant 0. \\ t & \text{otherwise.} \end{cases} \qquad \beta^*(y) = \begin{cases} \beta(y) & \text{if } f(\beta(y)) \leqslant 0. \\ s & \text{otherwise.} \end{cases}$$

That is, if $\alpha(x)$ (resp. $\beta(y)$) is an ancestor of t (resp. s) then x (resp. y) is mapped to the root t (resp. s). We note that

$$\begin{aligned} f(x) \leqslant g(\alpha^*(x)) \leqslant f(x) + \Delta, \\ g(y) \leqslant f(\beta^*(y)) \leqslant g(y) + \Delta. \end{aligned} \tag{4.4}$$

Indeed, if $\alpha^*(x) = \alpha(x)$ then $g(\alpha^*(x)) = f(x) + \Delta$. Otherwise $g(\alpha(x)) > 0$ and $g(\alpha^*(x)) = 0$. Since $f(x) \leqslant 0$, we obtain $g(\alpha^*(x)) < g(\alpha(x)) = f(x) + \Delta$. The same argument implies the second set of inequalities.

Consider the correspondence $\mathcal{C} \in T_1 \times T_2$ induced by α^* and β^* defined as:

$$\mathcal{C} := \{(x, \alpha^*(x)) \mid x \in T_1\} \cup \{(\beta^*(y), y) \mid y \in T_2\}.$$

We prove that $\mathrm{Dist}(\mathcal{C}) \leqslant 4\Delta$.

Indeed, consider any two pairs $(x_1, y_1), (x_2, y_2) \in \mathcal{C}$. Let u be the least common ancestor of x_1 and x_2 in T_1 (if T_1 is rooted at s), and w the least common ancestor of y_1 and y_2 in T_2 (if T_2 is rooted at t). Note that since T_1 and T_2 are trees, there is a unique path $x_1 \rightsquigarrow u \rightsquigarrow x_2$ between x_1 and x_2, such that $x_1 \rightsquigarrow u$ and $u \rightsquigarrow x_2$ are each monotone in function f values. This also implies that $d_1(x_1, u) = d_1(s, x_1) - d_1(s, u) = f(u) - f(x_1)$; similarly, $d_1(x_2, u) = f(u) - f(x_2)$. Symmetric statements hold for $y_1 \rightsquigarrow w \rightsquigarrow y_2$. Hence

$$d_1(x_1, x_2) = d_1(x_1, u) + d_1(u, x_2) = 2f(u) - f(x_1) - f(x_2),$$

$$d_2(y_1, y_2) = d_2(y_1, w) + d_2(w, y_2) = 2g(w) - g(y_1) - g(y_2).$$

We then have,

$$\begin{aligned}|d_1(x_1,x_2)-d_2(y_1,y_2)| &= |2f(u)-f(x_1)-f(x_2)-2g(w)+g(y_1)+g(y_2)|\\ &\leqslant 2|f(u)-g(w)|+|f(x_1)-g(y_1)|+|f(x_2)-g(y_2)|\\ &\leqslant 2|f(u)-g(w))|+2\Delta \text{ (by (4.4)).}\end{aligned}$$

We now wish to bound $|f(u)-g(w)|$. Consider the case where $y_1=\alpha^*(x_1), y_2=\alpha^*(x_2)$. As we traverse the path $x_1 \rightsquigarrow u \rightsquigarrow x_2$, the image of this path under α^* is a path $y_1 \rightsquigarrow \alpha^*(u) \rightsquigarrow y_2$, where $y_1 \rightsquigarrow \alpha^*(u)$ and $\alpha^*(u) \rightsquigarrow y_2$ are each monotone in function g values. Hence, $\alpha^*(u)$ is an ancestor of y_1 and y_2. Since w is the least common ancestor of y_1 and y_2, we have that $\alpha^*(u)$ must be an ancestor of w. The same claim can be similarly proved for the remaining cases. Further, it can be similarly shown that $\beta^*(w)$ must be an ancestor of u. Thus, $f(u)-\Delta \leqslant g(w) \leqslant f(u)+\Delta \Rightarrow |f(u)-g(w)| \leqslant \Delta$. We thus have

$$|d_1(x_1,x_2)-d_2(y_1,y_2)| \leqslant 4\Delta.$$

It then follows that $\text{Dist}(\mathcal{C}) \leqslant 4\Delta$. Since $d_{GH}(\mathcal{T}_1,\mathcal{T}_2) \leqslant \frac{1}{2}\text{Dist}(\mathcal{C})$, the lemma follows.

□

Next, we prove an upper bound on Δ.

Lemma 38. $\Delta \leqslant 14 d_{GH}(\mathcal{T}_1,\mathcal{T}_2)$.

Proof. Set $\delta = d_{GH}(\mathcal{T}_1,\mathcal{T}_2)$ and let $\mathcal{C}^* : T_1 \times T_2$ be an optimal correspondence that achieves $d_{GH}(\mathcal{T}_1,\mathcal{T}_2)$. Note that in general $d_{GH}(\mathcal{T}_1,\mathcal{T}_2)$ may only be achieved in the limit. In that case, our proof can be modified by considering a near-optimal correspondence of distortion $\delta = d_{GH}(\mathcal{T}_1,\mathcal{T}_2)+\varepsilon$ for some arbitrary $\varepsilon > 0$.

Let s be one of the endpoints of a longest simple path in T_1 (i.e, the length of this path realizes the diameter of $\mathcal{T}_1$); s is necessarily a leaf of T_1. Let (s,t) be a pair in $\mathcal{C}^*$. Consider the merge trees T_1^s and T_2^t defined by the functions f_s and f_t,

respectively. A result by Dey, Shi, and Wang [72, Corollary 6] (see also [73]) implies that

$$d_I(T_1^s, T_2^t) \leqslant 6\delta.$$

We prove below in Claim 39 that there is a vertex (in fact a leaf) $z \in V(T_2)$ such that $d_2(t, z) \leqslant 8\delta$.

It is easy to verify that

$$\|f_t - f_z\|_\infty \leqslant d_2(t, z) \leqslant 8\delta.$$

On the other hand, by the stability theorem of the interleaving distance,

$$d_I(T_2^t, T_2^z) \leqslant \|f_t - f_z\|_\infty \leqslant 8\delta.$$

By the triangle inequality,

$$\begin{aligned} d_I(T_1^s, T_2^z) \leqslant & d_I(T_1^s, T_2^t) + d_I(T_2^t, T_2^z) \\ \leqslant & 6\delta + 8\delta \\ \leqslant & 14\delta. \end{aligned}$$

This completes the proof of the lemma. □

Claim 39. *Let s be an endpoint of a longest simple path in T_1, and let (s, t) be a pair in $\mathcal{C}^*$. Then there is a vertex $z \in V(T_2)$ such that $d_2(t, z) \leqslant 8\delta$.*

Proof. Assume that there is no tree node within 8δ distance to t. In this case, t must be in the interior of an edge $e \in E(T_2)$. Let u_1 and u_2 be the two points in e from opposite sides of t such that $d_2(t, u_1) = d_2(t, u_2) = 8\delta + \nu$, where $\nu > 0$ is an arbitrarily small value. Both u_1 and u_2 exist, as there is no tree node of T_2 within 8δ distance to t, and

$$d_2(u_1, u_2) = d_2(t, u_1) + d_2(t, u_2) = 16\delta + 2\nu.$$

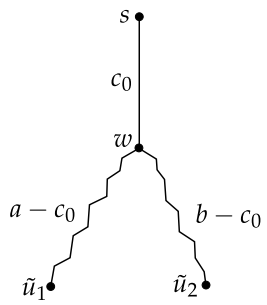


FIGURE 4.5: w is the nearest common ancestor of $\tilde{u}_1$ and $\tilde{u}_2$ in T_1^s.

Let $\tilde{u}_1, \tilde{u}_2 \in T_1$ be any corresponding points for u_1 and u_2 under $\mathcal{C}^*$, that is, $(\tilde{u}_1, u_1), (\tilde{u}_2, u_2) \in \mathcal{C}^*$. Since $\text{Dist}(\mathcal{C}^*) \leqslant 2\delta$, we have

$$d_1(\tilde{u}_1, \tilde{u}_2) \geqslant 14\delta + 2\nu. \tag{4.5}$$

On the other hand, since $d_2(t, u_1) = d_2(t, u_2) = 8\delta + \nu$, we have that

$$d_1(s, \tilde{u}_1), d_1(s, \tilde{u}_2) \in [6\delta + \nu, 10\delta + \nu]. \tag{4.6}$$

We now obtain an upper bound on $d_1(\tilde{u}_1, \tilde{u}_2)$.

If $\tilde{u}_1$ and $\tilde{u}_2$ have ancestor/descendant relation in T_1^s, then $d_1(\tilde{u}_1, \tilde{u}_2) = |d_1(s, \tilde{u}_1) - d_1(s, \tilde{u}_2)|$ and by (4.6), we thus have that $d_1(\tilde{u}_1, \tilde{u}_2) \leqslant 4\delta$, which contradicts (4.5). Otherwise, if $\tilde{u}_1$ and $\tilde{u}_2$ do not have ancestor/descendant relation, let w be the nearest common ancestor of $\tilde{u}_1$ and $\tilde{u}_2$ in T_1^s (see Figure 4.5). Let $c_0 = d_1(s, w)$. For simplicity, set $a = d_1(s, \tilde{u}_1)$ and $b = d_1(s, \tilde{u}_2)$. It then follows that

$$d_1(\tilde{u}_1, \tilde{u}_2) = a + b - 2c_0 \ . \ \text{Note,} \ \ a \geqslant c_0, b \geqslant c_0. \tag{4.7}$$

Since s is an endpoint of the longest path in T_1, it follows that $c_0 \geqslant \min\{a - c_0, b - c_0\}$. Indeed, if this were not the case, then without loss of generality, suppose the other point s' of the diameter pair is not in the subtree of T_1^s rooted at $\tilde{u}_1$. We then have

$$d_1(\tilde{u}_1, s') = (a - c_0) + (b - c_0) > c_0 + (b - c_0) = d_1(s, s'), \tag{4.8}$$

a contradiction. By (4.6), $a, b \geqslant 6\delta + \nu$. Thus

$$c_0 \geqslant \min\{a - c_0, b - c_0\} \geqslant 6\delta + \nu - c_0 \quad \Rightarrow c_0 \geqslant \frac{1}{2}(6\delta + \nu). \tag{4.9}$$

Combining (4.7) and (4.9), we have

$$d_1(\tilde{u}_1, \tilde{u}_2) \leqslant a + b - 6\delta - \nu \leqslant 20\delta + 2\nu - 6\delta - \nu = 14\delta + \nu, \tag{4.10}$$

contradicting (4.5). Thus, there exists $z \in V(T_2)$ such that $d_2(t, z) \leqslant 8\delta$. □

Remark. The proof of Claim 39 actually shows that t lies within distance 8δ of at least one leaf, as we never use the fact that u_1 and u_2 lie on the same edge of T_2. The only fact we use is that u_1 and u_2 lie on opposite sides of t at distance $8\delta + \nu$ each.

From Lemmas 37 and 38, we get the following.

Theorem 40. *Let* $\Delta = \min_{u \in V(T_1), v \in V(T_2)} d_I(T_1^u, T_2^v)$. *Then*

$$\tfrac{1}{2} d_{GH}(\mathcal{T}_1, \mathcal{T}_2) \leqslant \Delta \leqslant 14 d_{GH}(\mathcal{T}_1, \mathcal{T}_2).$$

In order to approximate $d_{GH}(\mathcal{T}_1, \mathcal{T}_2)$, we merely need to approximate the interleaving distance for each of the $O(n^2)$ pairs of merge trees obtained by rooting $\mathcal{T}_1$ and $\mathcal{T}_2$ at each of their vertices.

Corollary 41. *If there is a polynomial time, c-approximation algorithm for the interleaving distance between two merge trees, then there is a polynomial time,* $28c$*-approximation algorithm for the Gromov-Hausdorff distance between two metric trees that runs the algorithm for interleaving distance* $O(n^2)$ *times.*

4.5 Computing the Interleaving Distance

Let M_f and M_g be merge trees of two functions f and g, respectively. For simplicity, we use f and g to denote the height functions on M_f and M_g as well. Let n be the

total number of nodes in M_f and M_g, and let $r \geqslant 1$ be the ratio between the lengths of the longest and the shortest edges in M_f and M_g. We describe a $O(\min\{n, \sqrt{rn}\})$-approximation algorithm for computing $d_I(\mathsf{M}_f, \mathsf{M}_g)$.

Candidate values and binary search. We first show that a candidate set Λ of $O(n^2)$ values can be computed in $O(n^2)$ time such that $d_I(\mathsf{M}_f, \mathsf{M}_g) \in \Lambda$. Given Λ, we perform a binary search on Λ. At each step, we use a c-approximate decision procedure, for $c = c_1 \min\{n, \sqrt{rn}\}$ for some constant c_1, that given a value $\varepsilon > 0$ does the following : if $d_I(\mathsf{M}_f, \mathsf{M}_g) \leqslant \varepsilon$, it returns a pair of $c\varepsilon$-compatible maps between M_f and M_g; if $d_I(\mathsf{M}_f, \mathsf{M}_g) > \varepsilon$, it will either return a pair of $c\varepsilon$-compatible maps between M_f and M_g or report that no pair of ε-compatible maps exist.

We perform the binary search using the c-approximate decision procedure in the following way: if the procedure returns a pair of $c\varepsilon$-compatible maps for a value $\varepsilon \in \Lambda$, we continue the search using only lesser values of ε. Otherwise, we continue the search using only higher values of ε. When there are no more candidate values to search, we return the maps for the minimum value of ε that yielded a pair of maps.

The above procedure returns a pair of $c\varepsilon$-compatible maps where $d_I(\mathsf{M}_f, \mathsf{M}_g) \leqslant c\varepsilon \leqslant cd_I(\mathsf{M}_f, \mathsf{M}_g)$. Indeed, the procedure does not run out of candidate values $\varepsilon < d_I(\mathsf{M}_f, \mathsf{M}_g)$ until it has tried some $\varepsilon' \leqslant d_I(\mathsf{M}_f, \mathsf{M}_g)$ for which the approximate decision procedure returned a pair of $c\varepsilon'$-compatible maps. We now describe the candidate set Λ.

Let V_f (resp. V_g) be the set of nodes in M_f (resp. M_g). We define $\Lambda = \Lambda_{11} \cup \Lambda_{22} \cup \Lambda_{12}$, where

$$\Lambda_{11} = \{\tfrac{1}{2}|f(u) - f(v)| \mid u, v \in V_f\},$$
$$\Lambda_{22} = \{\tfrac{1}{2}|g(u) - g(v)| \mid u, v \in V_g\},$$
$$\Lambda_{12} = \{|f(u) - g(v)| \mid u \in V_f, v \in V_g\}.$$

Lemma 42. $d_I(\mathsf{M}_f, \mathsf{M}_g) \in \Lambda$.

Proof. Suppose to the contrary that $d_I(\mathsf{M}_f, \mathsf{M}_g) = \varepsilon \notin \Lambda$. Let $\alpha : \mathsf{M}_f \to \mathsf{M}_g$ and $\beta : \mathsf{M}_g \to \mathsf{M}_f$ be ε-compatible maps that realize $d_I(\mathsf{M}_f, \mathsf{M}_g) = \varepsilon$. We will obtain a contradiction by choosing $\varepsilon_0 > 0$ and constructing $(\varepsilon - \varepsilon_0)$-compatible maps $\hat{\alpha}, \hat{\beta}$.

For any point $x \in \mathsf{M}_f$, we define $\alpha_\downarrow(x) = \alpha(x)$ if $\alpha(x)$ is a node of M_g, otherwise $\alpha_\downarrow(x)$ is the lower endpoint of the edge of M_g containing $\alpha(x)$. Similarly we define the function $\beta_\downarrow : \mathsf{M}_g \to \mathsf{M}_f$.

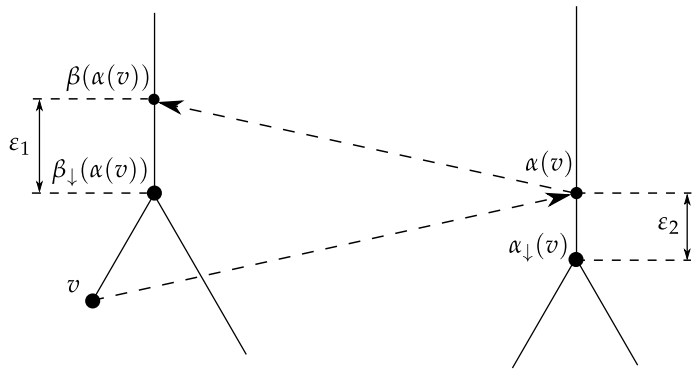


FIGURE 4.6: Trees M_f and M_g. Here $\delta_v = \min\{\frac{1}{2}\varepsilon_1, \varepsilon_2\}$.

For every node $v \in V_f$, $\alpha(v)$ (resp. $\beta(\alpha(v))$) lies in the interior of an edge of M_g (resp. M_f), because $\varepsilon \notin \Lambda \supseteq \Lambda_{12}$ (resp. Λ_{11}). We define

$$\delta_v = \min\{\tfrac{1}{2}\left(f(\beta(\alpha(v))) - f(\beta_\downarrow(\alpha(v))), g(\alpha(v)) - g(\alpha_\downarrow(v))\right)\}.$$

See Figure 4.6. Similarly we define δ_w for all $w \in V_g$. We set

$$\varepsilon_0 = \min\{\varepsilon, \min_{v \in V_f \cup V_g} \delta_v\}.$$

Since $\varepsilon \notin \Lambda$, we have $\varepsilon_0 > 0$. We now construct $(\varepsilon - \varepsilon_0)$-compatible maps $\hat{\alpha} : \mathsf{M}_f \to \mathsf{M}_g$ and $\hat{\beta} : \mathsf{M}_g \to \mathsf{M}_f$ (note $\varepsilon_0 \leqslant \varepsilon$, so $\varepsilon - \varepsilon_0$ is non-negative). We describe the construction of $\hat{\alpha}$; $\hat{\beta}$ is constructed similarly. By construction, for any node $u \in V_f$, $g(\alpha(u)) - g(\alpha_\downarrow(u)) \geqslant \varepsilon_0$, so we set $\hat{\alpha}(u)$ to be the point w on the edge of M_g containing

$\alpha(u)$ such that $g(w) = f(u)+\varepsilon-\varepsilon_0$. Once we have defined $\hat{\alpha}(u)$ and $\hat{\alpha}(v)$ for an edge $uv \in \mathsf{M}_f$, with $f(u) < f(v)$, we set $\hat{\alpha}(x)$, for a point $x \in uv$ with $f(x) = f(u) + \gamma$, to be

$$\hat{\alpha}(x) = \sigma_g^{\gamma}(\hat{\alpha}(u)).$$

That is, we set $\hat{\alpha}(x)$ to be the ancestor of $\hat{\alpha}(u)$ at height $f(u)+\varepsilon-\varepsilon_0+\gamma = f(x)+\varepsilon-\varepsilon_0$. Now, it's not too hard to see that if x_1 is an ancestor of x_2 in M_f, then $\hat{\alpha}(x_1)$ is an ancestor of $\hat{\alpha}(x_2)$ (similarly for $\hat{\beta}$). Further, $\hat{\beta}(\hat{\alpha}(x_1))$ is a descendant of $\beta(\alpha(x_1))$ for all $x_1 \in \mathsf{M}_f$ (a similar result holds for $\hat{\alpha} \circ \hat{\beta}$ and $\alpha \circ \beta$).

We claim that $\hat{\alpha}, \hat{\beta}$ are $(\varepsilon - \varepsilon_0)$-compatible. Indeed, by construction, $g(\hat{\alpha}(x)) = f(x) + \varepsilon - \varepsilon_0$ for all $x \in \mathsf{M}_f$, and $f(\hat{\beta}(y)) = g(y) + \varepsilon - \varepsilon_0$ for all $y \in \mathsf{M}_g$. We now prove that

$$\hat{\beta} \circ \hat{\alpha} = \sigma_f^{2(\varepsilon-\varepsilon_0)}.$$

Suppose to the contrary there is a point $x \in \mathsf{M}_f$ such that $y = \hat{\beta}(\hat{\alpha}(x)) \neq \sigma_f^{2(\varepsilon-\varepsilon_0)}(x)$.

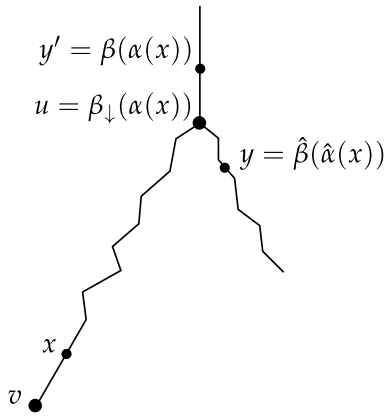


FIGURE 4.7: Figure showing u, v, x, y and y'.

Since $f(y) = f(x) + 2(\varepsilon - \varepsilon_0)$, y must not be an ancestor of x. On the other hand, α, β are ε-compatible, so $y' = \beta(\alpha(x))$ is the ancestor of x at height $f(x) + 2\varepsilon$. By construction of $\hat{\alpha}$ and $\hat{\beta}$, y is a descendant of y', in which case there is a node $u \in V_f$ that lies between y and y'. (If y and y' lie on the same edge of M_f, then y is also

an ancestor of x.) Let $u = \beta_{\downarrow}(\alpha(x))$. Let v be the lower endpoint of the edge e containing x. See Figure 4.7. Since $\varepsilon \notin \Lambda$, $f(u) \neq f(v) + 2\varepsilon$ (i.e., $u \neq \beta(\alpha(v))$). There are two cases to consider :

(i) $f(u) > f(v) + 2\varepsilon$. Then let $u = \beta(\alpha(z))$ for the point z lying between x and v at height $f(z) = f(u) - 2\varepsilon$. Furthermore $f(x) \geqslant f(z) > f(x) - 2\varepsilon_0$ (if $f(x) - f(z) \geqslant 2\varepsilon_0$, then $f(y') - f(u) = f(x) - f(z) \geqslant 2\varepsilon_0$, contradicting the fact that $f(y') - f(y) = 2\varepsilon_0$). Therefore we can choose a point $w \neq v$ on e such that $f(z) > f(w) > f(x) - 2\varepsilon_0$. We have $\hat{\beta}(\hat{\alpha}(w))$ is a descendant of $y = \hat{\beta}(\hat{\alpha}(x))$ (since w is a descendant of x). Moreover, $\beta(\alpha(w))$ is an ancestor of $\hat{\beta}(\hat{\alpha}(w))$. However, since $f(\beta(\alpha(w))) < f(u)$, $\beta(\alpha(w))$ lies between y and u. Thus, $\beta(\alpha(w))$ is not an ancestor of x (hence w), i.e., $\beta(\alpha(w)) \neq \sigma_f^{2\varepsilon}(w)$, contradicting the fact that α, β are ε-compatible.

(ii) $f(u) < f(v) + 2\varepsilon$. In this case

$$f(\beta(\alpha(v))) - f(\beta_{\downarrow}(\alpha(v))) \leqslant f(\beta(\alpha(v))) - f(u) < f(y') - f(y) = 2\varepsilon_0 \leqslant 2\delta_v,$$

which contradicts the definition of δ_v.

Hence we conclude that y is an ancestor of x, i.e., $\hat{\beta} \circ \hat{\alpha} = \sigma_f^{2(\varepsilon - \varepsilon_0)}$. Similarly, one can argue that $\hat{\alpha} \circ \hat{\beta} = \sigma_g^{2(\varepsilon - \varepsilon_0)}$, implying that $\hat{\alpha}, \hat{\beta}$ are $(\varepsilon - \varepsilon_0)$-compatible maps as claimed.

Putting everything together, we conclude that $\varepsilon \in \Lambda$. □

We now describe the decision procedure to answer the question "is $d_I(\mathsf{M}_f, \mathsf{M}_g) \leqslant \varepsilon$?" approximately. Given a parameter $\varepsilon > 0$, an edge is called ε-*long*, or *long* for brevity, if its length is strictly greater than 2ε. We first describe an exact decision procedure for the case when all edges in both trees are long, and then describe an approximate decision procedure for the case when the two trees have short edges.

Trees with long edges. We remove all degree-two nodes in the beginning. A *subtree* rooted at a point x in a merge tree M, denoted M^x, includes all the points in the merge tree that are descendants of x and an edge from x that extends upwards to height ∞. For a node $u \in V$, let $C(u)$ denote the children of u and let $p(u)$ denote its parent. Assume $d_I(\mathsf{M}_f, \mathsf{M}_g) \leqslant \varepsilon$, and let $\alpha : \mathsf{M}_f \to \mathsf{M}_g$ and $\beta : \mathsf{M}_g \to \mathsf{M}_f$ be a pair of ε-compatible maps. As in the proof of Lemma 42, we define the functions $\alpha_\downarrow$ and $\beta_\downarrow$ but restricted only to the vertices of M_f and M_g. That is, for a node $v \in V_f$, we define $\alpha_\downarrow(v)$ to be the lower endpoint of the edge containing $\alpha(v)$ – if $\alpha(v)$ is a node, then $\alpha_\downarrow(v)$ is $\alpha(v)$ itself. Similarly we define $\beta_\downarrow(w)$, for a node $w \in V_g$.

The following two properties of $\alpha_\downarrow$ and $\beta_\downarrow$ will be crucial for the decision procedure.

Lemma 43. *If all edges in M_f and M_g are ε-long, then the following hold: (i) For a node $v \in V_f$, $|f(v) - g(\alpha_\downarrow(v))| \leqslant \varepsilon$, and (ii) for a node $w \in V_g$, $|g(w) - f(\beta_\downarrow(w))| \leqslant \varepsilon$.*

Proof. We will prove part (i); part (ii) is similar. By definition, $g(\alpha_\downarrow(v)) \leqslant f(v) + \varepsilon$. Suppose $g(\alpha_\downarrow(v)) < f(v) - \varepsilon$. Let v' be a point in M_g lying on the edge containing $\alpha(v)$ and $\alpha_\downarrow(v)$ with height $f(v) - \varepsilon - \varepsilon_0$, for some sufficiently small ε_0. Then $\beta(v')$ lies in one of the subtrees rooted at the children of v, say M_1. Consider a descendant u of v at height $g(v') - \varepsilon$ lying in a different subtree M_2 rooted at v's child. Such a descendant exists, because all edges are ε-long. Since by definition and our choice of v' there does not exist any node in M_g between $\alpha(v)$ and v', we have $\alpha(u) = v'$. But then $\beta(\alpha(u)) = \beta(v')$ lies in M_1, and hence is not an ancestor of $u \in \mathsf{M}_2$; in other words $\beta(\alpha(u)) \neq \sigma_f^{2\varepsilon}(u)$. This contradicts the fact that α and β are ε-compatible. Thus, $g(\alpha_\downarrow(v)) \geqslant f(v) - \varepsilon$, and the claim follows. □

Lemma 44. *If all edges in M_f and M_g are ε-long, then $\alpha_\downarrow$ and $\beta_\downarrow$ are bijections with $\beta_\downarrow = \alpha_\downarrow^{-1}$ (and $\alpha_\downarrow = \beta_\downarrow^{-1}$).*

Proof. We will first show that $\beta_\downarrow = \alpha_\downarrow^{-1}$. Suppose to the contrary there exists a vertex $v \in V_f$ such that $\beta_\downarrow(\alpha_\downarrow(v)) = w \neq v$. Let $\alpha_\downarrow(v) = u$, for $u \in V_g$. From Lemma 43 we have $|f(v) - f(w)| \leqslant 2\varepsilon$. Since all edges are longer than 2ε and $v \neq w$, v cannot be an ancestor/descendant of w in M_f. By definition of $\alpha_\downarrow$, $\alpha(v)$ is an ancestor of $\alpha_\downarrow(v) = u$. Thus $\beta(\alpha(v))$ is an ancestor of $\beta(u)$. Further, $|f(v) - g(u)| \leqslant \varepsilon$ (Lemma 43) and $\beta(\alpha(v)) = \sigma_f^{2\varepsilon}(v)$ (since α, β are ε-compatible). Hence, $\beta(u)$ lies between v and $\beta(\alpha(v))$ on the edge e whose lower endpoint is v as e is ε-long. Thus, $\beta(u)$ is an ancestor of v. See Figure 4.8. Also by definition of $\beta_\downarrow$, $\beta(u)$ is an ancestor of $\beta_\downarrow(u) = w$. Thus, w is a descendant of v, a contradiction since v cannot be an ancestor of w.

We thus have $\beta_\downarrow = \alpha_\downarrow^{-1}$. Similarly, we can show that $\alpha_\downarrow = \beta_\downarrow^{-1}$. This also implies that $\alpha_\downarrow$ and $\beta_\downarrow$ are bijections. □

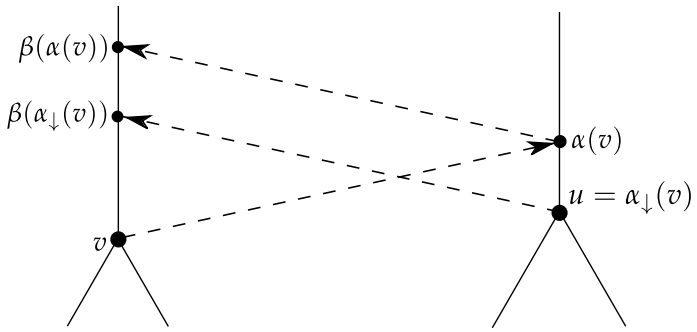


FIGURE 4.8: Figure showing $\beta(u) = \beta(\alpha_\downarrow(v))$ is an ancestor of v.

We define an indicator function $\Phi : V_f \times V_g \to \{0, 1\}$ such that

$$\Phi(u, v) = \begin{cases} 1, & \text{if } d_I(\mathsf{M}_f^u, \mathsf{M}_g^v) \leqslant \varepsilon, \\ 0, & \text{otherwise.} \end{cases}$$

The following lemma gives a recursive definition of $\Phi(u, v)$.

For two integers, k and ℓ with $k \leqslant \ell$, let $[k, \ell] = \{k, k+1, \ldots, \ell\}$.

Lemma 45. *Suppose all the edges in* M_f *and* M_g *are* ε*-long. Let* $(u, v) \in V_f \times V_g$*, and let* u_i *and* v_j *denote the* i*-th and* j*-th child, respectively, of* u *and* v*. We have* $\Phi(u, v) = 1$ *if and only if the following conditions hold : (i)* $|f(u) - g(v)| \leqslant \varepsilon$*, (ii)* $|C(u)| = |C(v)|$*, and (iii) there exists a permutation* π *of* $[1 : |C(u)|]$ *such that* $\Phi(u_i, v_{\pi(i)}) = 1$ *for all* $i \in [1 : |C(u)|]$*.*

Proof. Suppose $\Phi(u, v) = 1$, and let α, β be the corresponding ε-compatible maps. Suppose property (i) does not hold, and let $f(u) > g(v) + \varepsilon$ without loss of generality. Thus, $\beta(v)$ maps to one of the multiple edges incident to u, and there exists at least one edge $e = (u, w)$ with $w \in C(u)$ such that none of e's points (other than u) is in the image of β. However, $\beta(\alpha(w)) = \sigma_f^{2\varepsilon}(w)$ must lie in the interior of e (since e is ε-long), a contradiction. To prove that (ii) holds, note that by Lemma 44 and monotonicity of $\alpha_\downarrow$ and $\beta_\downarrow$, there exist bijections $\alpha_\downarrow, \beta_\downarrow$ between the vertices of M_f^u and M_g^v such that if $u_1 \in C(u_2)$ for a vertex $u_2 \in \mathsf{M}_f^u$, then $\alpha_\downarrow(u_1) \in C(\alpha_\downarrow(u_2))$ (a symmetric statement holds for $\beta_\downarrow$ and vertices in M_g^v). Thus, $\alpha_\downarrow, \beta_\downarrow$ induce bijections between $C(u)$ and $C(v)$, and hence $|C(u)| = |C(v)|$. Finally, for (iii), let $\alpha_\downarrow(u') = v'$ for some $u' \in C(u), v' \in C(v)$. Then by definition of $\alpha_\downarrow$ and $\beta_\downarrow$, $\alpha(\mathsf{M}_f^{u'}) \subseteq \mathsf{M}_g^{v'}$ and $\beta(\mathsf{M}_g^{v'}) \subseteq \mathsf{M}_f^{u'}$. This means that the restriction of the pair of ε-compatible maps α and β to $\mathsf{M}_f^{u'}$ and $\mathsf{M}_g^{v'}$ respectively remain ε-compatible for $\mathsf{M}_f^{u'}$ and $\mathsf{M}_g^{v'}$. Thus, $\Phi(u', v') = 1$, and the permutation π is defined by $\alpha_\downarrow, \beta_\downarrow$.

We now prove the opposite direction. Suppose properties (i),(ii) and (iii) hold. Let (α_i, β_i) be the pair of ε-compatible maps between $\mathsf{M}_f^{u_i}$ and $\mathsf{M}_g^{v_{\pi(i)}}$. Then, a pair of ε-compatible maps (α, β) between M_f^u and M_g^v is obtained as follows : $\alpha(x) = \{\alpha_i(x) \mid x \in \mathsf{M}_f^{u_i}\}$ (β is defined similarly). Note that points on the infinite edge from u (resp. v) upwards are *shared* among all $\mathsf{M}_f^{u_i}$ (resp. $\mathsf{M}_g^{v_j}$), whereas all other points in M_f^u (resp. M_g^v) are present in only one $\mathsf{M}_f^{u_i}$ (resp. $\mathsf{M}_g^{v_j}$). However, since $|f(u) - g(v)| \leqslant \varepsilon$, *shared* points are mapped to *shared* points and we have $|\alpha(x)| = 1$

(resp. $|\beta(y)| = 1|$) for all $x \in \mathsf{M}_f^u$ (resp. $y \in \mathsf{M}_g^v$). Thus, α and β are functions and satisfy all the required properties. Hence, $\Phi(u, v) = 1$. □

Decision procedure. We compute Φ for all pairs of nodes in $V_f \times V_g$ in a bottom-up manner and return $\Phi(r_f, r_g)$ where r_f (resp. r_g) is the root of M_f (resp. M_g). Let $(u, v) \in V_f \times V_g$.

Suppose we have computed $\Phi(u_i, v_j)$ for all $u_i \in C(u)$ and $v_j \in C(v)$. We compute $\Phi(u, v)$ as follows. If (i) or (ii) of Lemma 45 does not hold for u and v, then we return $\Phi(u, v) = 0$. Otherwise we construct the bipartite graph $G_{uv} = \{C(u) \cup C(v), E = \{(u_i, v_j) \mid \Phi(u_i, v_j) = 1\}\}$ and determine in $O(k^{5/2})$ time whether G_{uv} has a perfect matching, using the algorithm by Hopcroft and Karp [96]. Here, $k = |C(u)| = |C(v)|$. If G_{uv} has a perfect matching $M = \{(u_1, v_{\pi(1)}), \ldots, (u_k, v_{\pi(k)})\}$, we set $\Phi(u, v) = 1$, else we set $\Phi(u, v) = 0$. If $\Phi(u, v) = 1$, we use the ε-compatible maps for $\mathsf{M}_f^{u_i}, \mathsf{M}_g^{v_{\pi(i)}}$, for $1 \leqslant i \leqslant k$, to compute a pair of ε-compatible maps between M_f^u and M_g^v, as discussed in the proof of Lemma 45.

For a node $u \in V_f \cup V_g$, let k_u be the number of its children. The total time taken for running Hopcroft and Karp [96] is :

$$\sum_{u \in V(T_1)} \sum_{v \in V(T_2)} O\left(k_u k_v \sqrt{k_v}\right) = \sum_{u \in V(T_1)} k_u \sum_{v \in V(T_2)} O\left(k_v \sqrt{k_v}\right) \leqslant O\left(n^{3/2}\right) \sum_{u \in V(T_1)} k_u \leqslant O\left(n^{5/2}\right).$$

Hence we obtain the following.

Lemma 46. *Given two merge trees M_f and M_g and a parameter $\varepsilon > 0$ such that all edges of M_f and M_g are ε-long, then whether $d_I(\mathsf{M}_f, \mathsf{M}_g) \leqslant \varepsilon$ can be determined in $O(n^{5/2})$ time. If the answer is yes, a pair of ε-compatible maps between M_f and M_g can be computed within the same time.*

Trees with short edges. Given two merge trees, a naive map is to map the lowest among all the leaves in both the trees to a point at height equal to the height of the

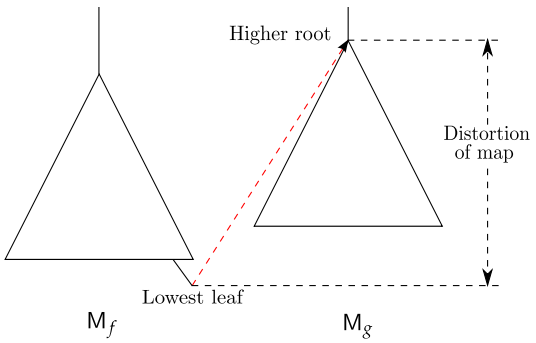


FIGURE 4.9: A naive map.

higher of the two roots of M_f and M_g (see Figure 4.9). Thus, all the points in one tree will be mapped to the infinitely long edge on the other tree. This map produces a distortion equal to the height of the trees, which can be arbitrarily larger than the optimum. Nevertheless, this simple idea leads to an approximation algorithm.

Here is an outline of the algorithm. After carefully *trimming* off short subtrees from the input trees, the algorithm decomposes the resulting trimmed trees into two kinds of regions – those with nodes and those without nodes. If the interleaving distance between the input trees is small, then there exists an isomorphism between trees induced by the regions without nodes. Using this isomorphism, the points in the nodeless regions are mapped without incurring additional distortion. Using a counting argument and the naive map described above, it is shown that the distortion incurred while mapping the regions with nodes and the trimmed regions is bounded.

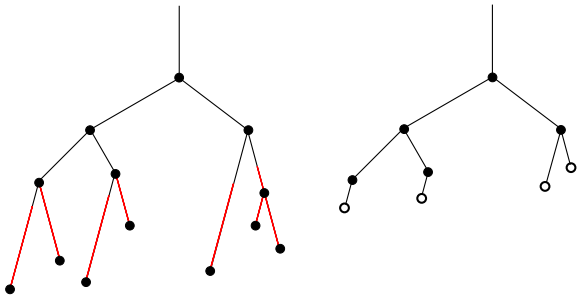


FIGURE 4.10: Trimming a tree : (left) original tree, red points have extent $< 2(\sqrt{2ns}+1)\varepsilon$; (right) trimmed tree, nodes added at the bottom (hollowed nodes).

More precisely, given M_f, M_g and $\varepsilon > 0$, define the *extent* $e(x)$ of a *point* x (which is not necessarily a tree node) in M_f or M_g as the maximum height difference between x and any of its descendants. Suppose each edge is at most $s\varepsilon$ long. Let M'_f and M'_g be subsets of M_f and M_g consisting only of points with extent at least $2(\sqrt{2ns}+1)\varepsilon$, adding nodes to the new leaves of M'_f and M'_g as necessary. Note that M'_f and M'_g themselves are trees, however they might contain nodes of degree 2. See Figure 4.10 for an example.

Lemma 47. *If* $d_I(\mathsf{M}_f, \mathsf{M}_g) \leqslant \varepsilon$, *then* $d_I(\mathsf{M}'_f, \mathsf{M}'_g) \leqslant \varepsilon$.

Proof. Let $\alpha : \mathsf{M}_f \to \mathsf{M}_g$ and $\beta : \mathsf{M}_g \to \mathsf{M}_f$ be ε-compatible maps. Let α' and β' be restrictions of the functions' domains to M'_f and M'_g respectively. We argue that the ranges of α' and β' lie in M'_g and M'_f respectively. Suppose otherwise. Then without loss of generality, there is a point $x \in \mathsf{M}'_f$ with $y = \alpha(x)$ not in M'_g. Because $x \in \mathsf{M}'_f$, its extent in M_f is at least $2(\sqrt{2ns}+1)\varepsilon$. Therefore, there exists a descendant x' of x in M_f with $f(x') = f(x) - 2(\sqrt{2ns}+1)\varepsilon$. Because y is not in M'_g, the extent of y must be less than $2(\sqrt{2ns}+1)\varepsilon$ and there exists no descendant y' of y with $g(y') = g(y) - 2(\sqrt{2ns}+1)\varepsilon = f(x) - 2(\sqrt{2ns}+1)\varepsilon + \varepsilon = f(x') + \varepsilon$. Since $g(\alpha(x')) = f(x') + \varepsilon$, $\alpha(x')$ is not a descendant of $\alpha(x)$, which contradicts the assumption that α, β are ε-compatible maps. □

The above lemma can be easily generalized to say that removing points in both trees with extent less than or equal to any fixed value does not increase the distance between them.

We now define *matching points* in M'_f and M'_g. Let H be the set of function values for leaves and branching nodes in M'_f or M'_g, and let $H' \subseteq H$ be the subset of H consisting of function values h for which $(h, h+2\varepsilon]$ does not intersect H. A point x in M'_f is a matching point if $f(x) \in H'$. Similarly, a point y in M'_g is a matching point

if $g(y) \in H'$. By this definition, no two matching points share a function value within 2ε of each other unless they share the exact same function value. Furthermore, if x is a matching point, then all points with the same function value as x on both M'_f and M'_g are matching points. There are $O(n^2)$ matching points.

Suppose $d_I(\mathsf{M}'_f, \mathsf{M}'_g) \leqslant \varepsilon$, and let $\alpha' : \mathsf{M}'_f \to \mathsf{M}'_g$ and $\beta' : \mathsf{M}'_g \to \mathsf{M}'_f$ be a pair of ε-compatible maps. Call a matching point x in M'_f and a matching point y in M'_g with $f(x) = g(y)$ *matched* if $\alpha'(x)$ is an ancestor of y.

Lemma 48. *Let x be any matching point in M'_f. The matched relation between matching points in M'_f at height $f(x)$ and matching points in M'_g at height $f(x)$ is a bijective function.*

Proof. No two distinct matching points y_1 and y_2 on M'_g with $f(x) = g(y_1) = g(y_2)$ share the same ancestor with function value $f(x)+\varepsilon$, because they have no branching node ancestors with low enough function value. Therefore, a matching point in M'_f can be matched to only one matching point in M'_g.

Let x_1 and x_2 be two distinct matching points from M'_f with $f(x) = f(x_1) = f(x_2)$. If $\alpha'(x_1)$ and $\alpha'(x_2)$ are ancestors of a common matching point y, then $\alpha'(x_1) = \alpha'(x_2)$ and thus x_1 and x_2 must have a common ancestor x' at height $f(x) + 2\varepsilon$. However, x_1 and x_2 have no branching node ancestor with low enough function value for x' to exist. Hence, the matching relation must be injective from matching points in M'_f to M'_g.

Finally, consider any matching point y on M'_g with $g(y) = f(x)$. Point $x'_1 = \beta'(y)$ is the ancestor of a matching point x_1 on M'_f with $f(x_1) \geqslant g(y)$. (Note that by the same argument as the beginning of this proof, only one such matching point x_1 can exist.) Point $y' = \alpha'(x'_1)$ is an ancestor of y with $g(y') \leqslant g(y) + 2\varepsilon$. Point y is the only descendant of y' with function value $f(x)$. Point $\alpha'(x_1)$ must be an ancestor of y, meaning x_1 and y are matched. Thus, the matching relation is surjective. □

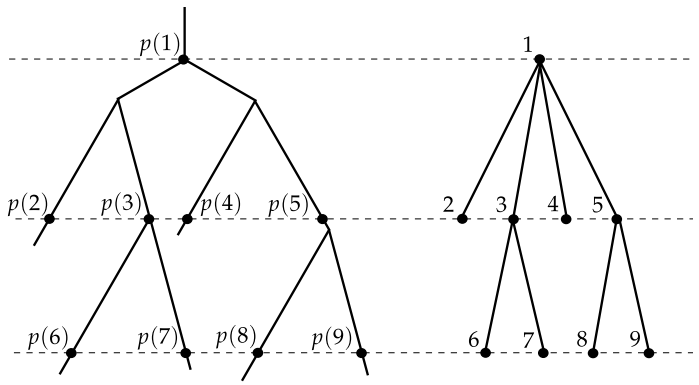


FIGURE 4.11: The left tree shows matching points on tree M'_f and the right tree shows $\tilde{\mathsf{M}}_f$.

We now define a rooted tree $\tilde{\mathsf{M}}_f$ to be a rooted tree consisting of one node per matching point on M'_f. Let $p(v)$ be the matching point for node v. $\tilde{\mathsf{M}}_f$ has node v as an ancestor of node u if $p(v)$ is an ancestor of $p(u)$ (see Figure 4.11). Define $\tilde{\mathsf{M}}_g$ similarly. The size of $\tilde{\mathsf{M}}_f$ and $\tilde{\mathsf{M}}_g$ is $O(n^2)$. Intuitively, $\tilde{\mathsf{M}}_f$ and $\tilde{\mathsf{M}}_g$ represent the trees induced by matching points. By the definition of interleaving distance and Lemma 48, $\tilde{\mathsf{M}}_f$ and $\tilde{\mathsf{M}}_g$ are isomorphic if M'_f and M'_g satisfy that $d_I(\mathsf{M}'_f, \mathsf{M}'_g) \leqslant \varepsilon$.

Decision procedure. We are now ready to describe the decision procedure. We first construct the subtrees M'_f and M'_g of M_f and M_g, respectively, consisting of points with extent at least $2(\sqrt{2ns}+1)\varepsilon$. Next, we compute matching points on M'_f and M'_g and construct the trees $\tilde{\mathsf{M}}_f$ and $\tilde{\mathsf{M}}_g$ on these matching points, as defined above.

Using the algorithm of [24, chap. 3, p. 85], we determine in time linear in the size of the trees whether $\tilde{\mathsf{M}}_f$ and $\tilde{\mathsf{M}}_g$ are isomorphic. If the answer is no, we return no. By Lemma 48, $d_I(\mathsf{M}_f, \mathsf{M}_g) > \varepsilon$ in this case. Otherwise we construct the following functions $\alpha : \mathsf{M}_f \to \mathsf{M}_g$ and $\beta : \mathsf{M}_g \to \mathsf{M}_f$ and return them. Recall, it suffices to perform assignments where the function value increases by *at most* $c\varepsilon$. For each pair of matching points x and y matched by the isomorphism, the algorithm sets $\alpha(x) = y$ and $\beta(y) = x$. Now, let (ξ_1, ξ_2) be any maximal range of function values without any

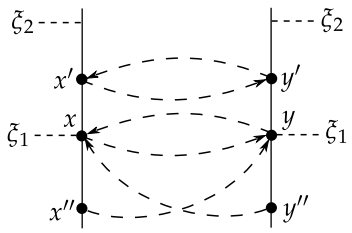


FIGURE 4.12: Figure showing points x, y, x', y', x'', y''.

branching nodes or leaves in M'_f or M'_g with $\xi_2 - \xi_1 > 2\varepsilon$. Let x' be any point in M'_f with $f(x') \in (\xi_1, \xi_2)$. Point x' has a unique matching point descendant x at height ξ_1, by the definition of matching points. The algorithm sets $\alpha(x')$ to the point y' in M'_g where y' is the ancestor of $\alpha(x)$ with $g(y') = f(x')$, and it sets $\beta(y') = x'$. For every remaining point x'' in M'_f, the algorithm sets $\alpha(x'')$ to $\alpha(x)$ where x is the lowest matching point *ancestor* of x''. $\beta(y'')$ is defined similarly for remaining points y'' in M'_g that were not paired with some x'. We call such points x'' and y'' *lazily assigned*. See Figure 4.12. Finally, each point z in $\mathsf{M}_f - \mathsf{M}'_f$ has $\alpha(z)$ set to $\alpha(x)$ where x is the lowest ancestor of z in M'_f. Similar assignments are done for points in $\mathsf{M}_g - \mathsf{M}'_g$.

Lemma 49. *(i) For each lazily assigned point $x'' \in \mathsf{M}'_f$,*

$$g(\alpha(x'')) \leqslant f(x'') + 2(\sqrt{2ns} + 1)\varepsilon.$$

(ii) For each lazily assigned point $y'' \in \mathsf{M}'_g$,

$$f(\beta(y'')) \leqslant g(y'') + 2(\sqrt{2ns} + 1)\varepsilon.$$

Proof. We only prove (i); (ii) is symmetric. The higher of the two roots of M'_f and M'_g is a matching point, and so are all the points at that height. Thus, all lazily assigned points have a matching point ancestor. We show that the nearest such ancestor cannot be too much higher up.

Let x be a matching point. We show that there exists a region (ξ_1, ξ_2) as defined above with

$$f(x) - 2(\sqrt{2ns} + 1)\varepsilon \leqslant \xi_2 \leqslant f(x).$$

Consider sweeping over the function values downward starting at $f(x)$ and let ξ_2 be the largest function value possible for a region as defined above. If the sweep line ever goes a distance greater than 2ε without encountering a branching node or leaf in M'_f or M'_g, then an ξ_2 is found. Therefore, there will be at least one branching node or leaf x' in M'_f or M'_g per descent of 2ε until ξ_2 is found. Suppose $\xi_2 < f(x) - 2(\sqrt{2ns}+1)\varepsilon$. Let $l = \sqrt{2ns} + 1$, and $f' = f(x) - 2l\varepsilon$.

Let $\{P_1, P_2, \dots\}$ be a set of paths from each branching node of M'_f or M'_g encountered during the sweep to leaves of M_f and M_g such that for each pair of paths, both paths are disjoint except possibly at the higher endpoint of one of the two paths. Such a set of paths can be found by greedily selecting an arbitrary path for each branching node as it is encountered. In addition, let $\{Q_1, Q_2, \dots\}$ be a set of pairwise-disjoint paths from leaves of M'_f and M'_g to leaves of M_f and M_g. Because each point in M'_f and M'_g has extent at least $2l\varepsilon$, the lower endpoint of each of these paths lies below height f'. Since edge lengths are at most $s\varepsilon$, there are at least $(f(x') - f')/s\varepsilon$ nodes in M_f or M_g on the path P_i or Q_j from each branching node or leaf x' at height $f(x') \in [f', f(x))$, *not counting x' itself.* In total, these paths contain at least

$$\sum_{i=1}^{l} \frac{(l-i)2\varepsilon}{s\varepsilon} = \frac{l(l-1)}{s}$$

nodes, not counting their higher endpoints. Each node counted above appears on at most one path P_i and at most one path Q_j, for at most two paths total, so

$$\frac{l(l-1)}{s} \leqslant 2n \Rightarrow l(l-1) \leqslant 2ns,$$

a contradiction since $l = \sqrt{2ns} + 1$. Therefore, either $\xi_2 \geqslant f(x) - 2(\sqrt{2ns}+1)\varepsilon$, or the trees M'_f and M'_g do not extend below height $f(x) - 2(\sqrt{2ns}+1)\varepsilon$. In either case, the lemma follows. □

Lemma 50. *Let M_f and M_g be two merge trees and let $\varepsilon > 0$ be a parameter. There is an $O(n^2)$ time algorithm that returns a pair of $4(\sqrt{2ns}+1)\varepsilon$-compatible maps between M_f and M_g, if $d_I(\mathsf{M}_f, \mathsf{M}_g) \leqslant \varepsilon$ and the maximum length of a tree edge is $s\varepsilon$. If $d_I(\mathsf{M}_f, \mathsf{M}_g) > \varepsilon$, then the algorithm may return no or return a pair of $4(\sqrt{2ns}+1)\varepsilon$-compatible maps.*

Proof. Constructing the trees $\tilde{\mathsf{M}}_f$ and $\tilde{\mathsf{M}}_g$, the corresponding isomorphism between them (if it exists), and the maps α and β between M_f and M_g (if they exist) takes time $O(n^2)$.

Except for the lazily assigned points, all the points in M'_f and M'_g are mapped by α and β resp. to points at the same function value. By Lemma 49, each point in M'_f and M'_g has its function value changed by at most $2(\sqrt{2ns}+1)\varepsilon$. Points in $\mathsf{M}_f - \mathsf{M}'_f$ (resp. $\mathsf{M}_g - \mathsf{M}'_g$) have their nearest ancestors in M'_f (resp. M'_g) at function value at most $2(\sqrt{2ns}+1)\varepsilon$ away. Since α and β map them to the images of their nearest ancestors, their function values change by at most $2 \cdot 2(\sqrt{2ns}+1)\varepsilon$. □

Remark. (i) Since the minimum edge length is $\leqslant 2\varepsilon$, the maximum edge length is $s\varepsilon$, and the ratio between the lengths of the longest and shortest edges is r; we have $r \geqslant s/2$.

(ii) If $s = \Omega(n)$, we modify the above algorithm slightly – we skip the trimming step, but keep the rest same. It can be shown, as in Lemma 49, that the height of a point and its image differ by at most $2n\varepsilon$. In particular, the proof no longer requires as complicated a counting argument, because any path contains at most n nodes.

Putting it together. By Lemmas 46 and 50, the decision procedure takes $O(n^{5/2})$ time. If it returns no, then $d_I(\mathsf{M}_f, \mathsf{M}_g) > \varepsilon$. If it returns yes, then it also returns $O(\min\{n, \sqrt{rn}\}\varepsilon)$-compatible maps between them. Hence, we conclude the following.

Theorem 51. *Given two merge trees M_f and M_g with a total of n vertices, there exists an $O(n^{5/2}\log n)$-time algorithm with an approximation factor of $O\left(\min\{n, \sqrt{rn}\}\right)$ for computing the interleaving distance between them, where r is the ratio between the lengths of the longest and the shortest edge in both trees.*

Combining Theorem 51 with Corollary 41, we have:

Corollary 52. *Given two metric trees $\mathcal{T}_1$ and $\mathcal{T}_2$ with a total of n vertices, there exists an $O(n^{9/2}\log n)$-time algorithm with an approximation factor of $O\left(\min\{n, \sqrt{rn}\}\right)$ for computing the Gromov-Hausdorff distance between them, where r is the ratio between the lengths of the longest and the shortest edge in both trees.*

4.6 Conclusion

We have presented the first hardness results for computing the Gromov-Hausdorff distance between metric trees. We have also given a polynomial time approximation algorithm for the problem. But the current gap between the lower and upper bounds on the approximation factor is polynomially large. While we would like to reduce this gap, doing so seems very difficult. On the algorithmic side of things in particular, trying for anything less than an $O(\sqrt{n})$-approximation appears to prevent our use of algorithms for graph isomorphism, the strongest algorithmic tool used in the above algorithm. We hope that our current investigation will stimulate more research on the theoretical and algorithmic aspects of embedding or matching under additive metric distortion.

5

Subtrajectory Clustering

5.1 Introduction

Trajectories arise in the description of any dynamic physical system. They are being recorded or inferred from millions of sensors at an unprecedented scale. To take full benefit of the enormous opportunities provided by these datasets for extracting useful information and improving decision making, several computational challenges need to be addressed. This chapter focuses on one such computational problem, namely extracting high-level shared structure that encodes much of the information present in a large trajectory dataset. For specificity, we focus on GPS traces of moving objects, though much of the work here can be extended to many other domains. We assume that each trajectory is observed as a sequence of points. For simplicity, we refer to the point sequences themselves as trajectories. For such trajectories, common but unknown constraints and objectives generate patterns of *subtrajectories*, portions of trajectories that are commonly shared. See Figure 5.1 for an example. Our goal is to cluster such shared subtrajectories and represent each cluster as a *pathlet*, a sequence of points that is not necessarily a subsequence of an input trajectory. For

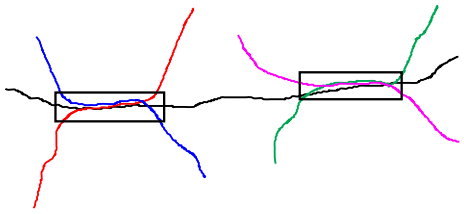


FIGURE 5.1: The middle black trajectory shares commonalities with different sets of trajectories, as shown in the boxed regions. The subtrajectories inside these boxes can be clustered together, and a representative pathlet for each cluster computed so that the black trajectory can be (almost) obtained by a concatenation of these pathlets, with gaps in between.

example, a road network contributes to shared subtrajectories for vehicle trajectories. Intuitively, a pathlet is a portion of a road/path that tends to be traversed *as a whole* by many trajectories. The *subtrajectory clustering* problem is to partition shared subtrajectories into a small number of clusters such that the pathlet for each cluster is a high-quality representation of subtrajectories in each cluster.

Subtrajectory clustering is an interesting trajectory segmentation problem from both a modeling and algorithmic standpoint. Individual trajectories carry little information about shared structures, and it is only in the context of a collection of trajectories that the importance of certain subtrajectories becomes evident. This situation is different from some other cases where shared structures (motifs) are extracted from a set of curves, such as protein backbones or speech signals. In both of these cases, motifs are more structured, and domain knowledge provides enough information to identify common substructures in each curve, such as secondary structures (α-helices and β-sheets) in a protein backbone.

If trajectories correspond to traffic on a known road network, then these trajectories can be mapped to paths on a graph that models the road network. However, in many cases there might not be an underlying road network or it might not be known, e.g., trajectories corresponding to pedestrian movement in an urban environment or

pixels in video data. Therefore, we focus on unmapped trajectory point sequences in $\mathbb{R}^2$ sampled at discrete time stamps.

Not only do pathlets provide a compression of large trajectory data, effectively performing non-linear dimension reduction, the semantic information provided by pathlets have been useful for trajectory analysis applications such as similarity search and anomaly detection [137]. Furthermore, a pathlet representation of trajectories decreases uncertainty in individual trajectories; it reduces noise, fills missing data, and identifies outliers (see Figure 5.8) [109, 110]. Pathlets have also been used for many other applications such as reconstructing a road network from trajectory data [49, 109].

5.1.1 Our contribution

Following are our three main contributions – a simple model for subtrajectory clustering; efficient approximation algorithms with provable guarantees on the quality of solution and running time under this model; and experimental results that demonstrate the efficacy and efficiency of our model and algorithms, respectively.

Modeling. Our first main contribution, given in Section 5.2, is a simple model for subtrajectory clustering. In our model, each cluster of subtrajectories is represented as a *pathlet*, a sequence of points. A subtrajectory clustering is a set $\mathcal{P}$ of pathlets along with an assignment of subtrajectories to pathlets in $\mathcal{P}$. We use a distance function between two point sequences (e.g., discrete Fréchet distance [33]) to measure how well a pathlet represents (covers) a subtrajectory.

Such a model is faithful to the following three desiderata. As discussed in Section 5.1.2, though related work has considered some of these desiderata, we believe we are the first to address all of these simultaneously.

Robustness to variations. We allow trajectories to have *gaps*, i.e., we do not

require that the entire trajectory is covered by pathlets. This serves two purposes – in addition to handling noisy portions of trajectories better (akin to how gaps in DNA sequence alignment model mutations [57]), these gaps also model those portions of a trajectory that may be unique to it and not shared with other trajectories (see Figure 5.1 and Section 5.7).

Theoretical guarantees. We define a single objective function that is a weighted sum of the number of pathlets chosen, the cluster quality for each pathlet, and the gap penalty for each trajectory. By varying weights, one can obtain a trade-off between the three criteria. An optimal subtrajectory clustering that minimizes the objective should result in a small number of good quality clusters (pathlets) and few gaps. Further, one should be able to cover shared portions of an input trajectory by a short sequence of pathlets (see Figure 5.9). As we show, such an objective admits to efficient approximation algorithms.

Data-driven clustering. Given the above objective, the number of pathlets chosen, the subtrajectory assignments, and the gaps in the trajectories are all decided by the algorithm in a data-driven fashion, largely independent of any user input. Further, we assume that the set $\mathbb{P}$ of candidate pathlets from which the set $\mathcal{P}$ of pathlets is constructed is the set of all possible point sequences, and we do not make any *a priori* restrictions on what the pathlets can look like. In this sense, our model is entirely *data-driven*.

In the subtrajectory-clustering problem, we assume that the candidate pathlets is the set of all possible point sequences. We also consider the variant in which $\mathcal{P}$ is chosen from a given finite set $\mathbb{P}$ of b pathlets. We refer to this variant as the *pathlet-cover* problem. Besides being interesting in its own right, our subtrajectory-clustering algorithm will need an algorithm for this problem. For both variants, we are given a set $\mathcal{T}$ of n trajectories containing m points in total.

Algorithms. We show (Section 5.3) that both pathlet cover and subtrajectory clustering are NP-hard by a simple reduction from the standard Facility-Location problem in Euclidean space.

We then turn to our main algorithmic results – approximation algorithms for pathlet cover and subtrajectory clustering. Our algorithm for pathlet-cover (Section 5.4) finds an $O(\log m)$-approximate pathlet dictionary in $\tilde{O}(bm^3)$ time[1]. To obtain this result, we show that the construction of $\mathcal{P}$ can be formulated as an instance of the weighted Set-Cover problem so that a greedy algorithm can be used to construct a pathlet dictionary of the appropriate cost. However, the size of the set system can be as large as $\Omega(2^n)$. The main technical contribution is to show how the greedy algorithm can be executed efficiently without constructing the set system explicitly; a similar idea was employed in inventory management [112].

Next we describe approximation algorithms for subtrajectory clustering (Section 5.5). Here we use discrete Fréchet distance, a widely used method for measuring similarity between two point sequences, to measure the quality of a pathlet covering a subtrajectory; see Section 5.5 for the formal definition. By a simple reduction to the pathlet-cover problem, we obtain a polynomial-time $O(\log m)$-approximation. We simply use the set of all $O(m^2)$ contiguous subtrajectories of input trajectories as the candidate set $\mathbb{P}$. However, such a "brute-force" enumeration of candidate pathlets makes the running time prohibitive. In essence, this algorithm considers too many possible assignments between subtrajectories and candidate pathlets, and neither takes full advantage of the freedom in our choice of pathlets nor does it exploit the underlying geometry fully.

We first show via a geometric grouping argument that by paying an extra $\log m$

[1] We use the notation $\tilde{O}$ to hide polylogarithmic factors in n, m, b, and σ, the *spread* of the trajectories' points. The spread of a point set is defined as the ratio of largest to smallest pairwise distance. The exponent on the polylog may depend upon the dimension d of the underlying space when our algorithms are generalized beyond $\mathbb{R}^2$.

factor in the approximation quality, we can work with a set $\mathbb{P}$ of $O(m)$ candidate pathlets. We then describe how to expedite the algorithm further and prove that if the input trajectories are κ-packed [2] for a constant $\kappa > 0$, then the algorithm runs in $\tilde{O}(m^2)$ time. Intuitively, a trajectory is κ-packed if it is not a space-filling curve or a fractal. In practice, GPS trajectories are κ-packed with small values of κ; see Section 5.7 and [147]. The main technical result, which is quite surprising, is to show that for κ-packed curves for constant κ, the number of subtrajectories of any trajectory that can be mapped to a single pathlet is $\tilde{O}(1)$; this pruning of subtrajectories also leads to a more robust solution. By use of appropriate data structures, the corresponding steps of the greedy algorithm can therefore be significantly speeded up.

Preprocessing and experiments. We briefly discuss two additional preprocessing steps that improve our algorithms' performance in practice (Section 5.6). The first step is designed to handle lossy or sparse data sets for which many trajectories are only partially observed. We describe a method to fill in the missing observations not through a simple linear interpolation, but instead using observations from other input trajectories. The second step is designed to even further reduce the size of the set $\mathbb{P}$ of candidate pathlets. Our method sparsifies the set $\mathbb{P}$ by first projecting them to an Euclidean space and then applying a clustering technique to remove redundant candidate pathlets.

We conclude with an empirical evaluation of our algorithms and the properties of our model in Section 5.7. We perform our experiments on both synthetic and real data sets, and show that the algorithm handles the desiderata of being robust to variations, being efficient and accurate, and being data-driven. In particular, it

[2] A trajectory is κ-packed if the length of its intersection with any disk of radius r is at most κr. See Section 5.5 for details.

finds pathlets that cover the dense areas of the data sets, and individual trajectories can be recovered from a subset of the pathlets with few gaps that capture variations in individual trajectories. We also show that meaningful pathlets can be found even on instances where there is no underlying road network, and further, the resulting pathlets gracefully handle noise in the individual trajectories. In addition to examining the quality of our algorithm's output visually, we also demonstrate that the parameters can be tuned for different balances between dictionary size, coverage of trajectories, and similarity between pathlets and their assigned subtrajectories.

5.1.2 Related work

Though there has been a recent line of work on subtrajectory clustering (see Section 1.3), all of this only considers a subset of the desiderata we consider. In particular, none of these algorithms provide any guarantee on their performance in the worst case. In contrast, our goal was to develop an algorithm with provable guarantees on its performance. Further, either the works aim for complete coverage without the notion of gaps, or are not data-driven, imposing particular structure on either the trajectories or pathlets. In some more detail, the algorithm in Chen *et al.* [62] assumes the trajectories to be paths in a graph with no noise and they do not consider gaps. Though the approach in Panagiotakis *et al.* [128] appears superficially similar to ours, their method segments trajectories based on the representativeness of individual trajectory edges, which is based on density near the edge (irrespective of which trajectories contribute to the density). The algorithm in Lee *et al.* [106] requires pathlets to be line segments, and the algorithm in Sankararaman *et al.* [134] is effective only if data is dense, not too noisy, and not too big. The algorithm in Buchin *et al.* [50] requires the user to specify two of the following three parameters – size, length or diameter of the cluster; their algorithm finds the single cluster that optimizes for the third. Their approach is slightly modified in [49] to detect multiple

subtrajectory clusters from trajectories; however they assume the trajectories are points sampled from an unknown, underlying road network; and they do not have a notion of gaps. In summary, our work encompasses the desiderata that previous work either partially covers or omits entirely.

Although there is a fair bit of work on extracting common movement patterns as discussed earlier (Section 1.3), this line of work either pre-specifies the portion of trajectories where to find a pattern or pre-specifies the pattern. In contrast, our goal is to identify shared structures determined completely by the data, while being robust to noise, missing data, and non-uniform sampling.

The setting for multiple sequence alignment in computational biology [127] is much simpler, as it deals with one-dimensional sequences over a finite alphabet. The work on functional clustering [133] assumes a (known) global parametrization over the data, and the range of functions is one dimensional which makes the problem simpler. Topic modeling/dictionary learning [44, 76] if applied to trajectory data will not return subtrajectories as "topics" or words in the dictionary, as they do not concern themselves with locality.

5.2 The Clustering Model

A trajectory or pathlet is a polygonal curve defined by a finite sequence of points $\langle p_1, p_2, \ldots \rangle$ in $\mathbb{R}^2$ (again, our results can be generalized to trajectories in $\mathbb{R}^d$ for any constant d). Let $\mathcal{T} = \{T_1, \ldots, T_n\}$ be a set of n trajectories and $\mathbb{P} = \{P_1, \ldots, P_b\}$ be a set of b candidate pathlets. For simplicity, we assume the points in each trajectory are distinct and let $\mathbb{X} = \bigcup_{i=1}^{n} T_i$ be the set of all trajectory points. Set $m = |\mathbb{X}|$. Let $T[p, q]$ denote the subtrajectory of T lying between points p and q of T. For positive integers i, j, let $T(i, j)$ denote $T[p_i, p_j]$.

A subtrajectory *clustering* is a *pathlet dictionary*, that is, a (multi) subset $\mathcal{P} \subseteq \mathbb{P}$, along with an assignment of a subset $\mathcal{T}(P)$ of subtrajectories to each $P \in \mathcal{P}$ such

that there is at most one subtrajectory $S \in \mathcal{T}(P)$ of each trajectory $T \in \mathcal{T}$. In turn, for each trajectory $T \in \mathcal{T}$, we let $T_P \in \mathcal{T}(P)$ denote the subtrajectory of T assigned to P, assuming one exists. We say that each $S \in \mathcal{T}(P)$ is *assigned to* or *covered by* P. We want to compute a multi set of pathlets $\mathcal{P}$ (and their assignments)[3] that is succinct and captures the shared portions between trajectories.

In our model, a good clustering minimizes an objective function consisting of three terms. The first term is proportional to $|\mathcal{P}|$, the number of pathlets. The second term consists of the fraction of points of each trajectory that are not assigned to any pathlet in $\mathcal{P}$, summed over all trajectories in $\mathcal{T}$. In other words, the second term measures the size of the *gaps* left uncovered by $\mathcal{P}$. We focus on the fraction of uncovered points in each trajectory instead of the absolute number, because we do not wish to optimize only for a small number of especially long or densely sampled trajectories. The third term captures how well the assigned subtrajectories resemble the pathlets in the dictionary.

To formally define the third term in our objective, we use a distance function, denoted by $\mathtt{d}(T_1, T_2)$, to measure the distance between two point sequences T_1 and T_2. To eliminate the possibility of certain assignments, we may have $\mathtt{d}(S, P) = \infty$ in some cases. We say an assignment is *permissible* if $\mathtt{d}(S, P) \neq \infty$.

For a trajectory $T \in \mathcal{T}$, let $\tau(T)$ be the fraction of the trajectory's points that is not covered any pathlet. The cost of covering $\mathcal{T}$ by $\mathcal{P}$ is defined as:

$$\mu(\mathcal{T}, \mathcal{P}, \mathtt{d}) = c_1|\mathcal{P}| + c_2 \sum_{T \in \mathcal{T}} \tau(T) + c_3 \sum_{P \in \mathcal{P}} \sum_{S \in \mathcal{T}(P)} \mathtt{d}(S, P),$$

where c_1, c_2 and c_3 are user-defined parameters.

Given a tuple $(\mathcal{T}, \mathbb{P}, \mathtt{d})$, the *pathlet-cover problem* is to compute $\mathcal{P}^* \subseteq \mathbb{P}$ and

[3] We allow $\mathcal{P}$ to contain the same point sequence multiple times so that each copy may be assigned to different portions of a single trajectory.

permissible assignments of subtrajectories to pathlets, to minimize $\mu(\mathcal{T}, \mathcal{P}, \mathtt{d})$. We let $\pi(\mathcal{T}, \mathbb{P}, \mathtt{d}) = \mu(\mathcal{T}, \mathcal{P}^*, \mathtt{d})$.

The *subtrajectory-clustering problem* is to solve the pathlet-cover instance $(\mathcal{T}, \mathbb{P}, \mathtt{d})$ with $\mathbb{P}$ being the (uncountably infinite) set of all point sequences and $\mathtt{d}$ again being an arbitrary distance function between point sequences.

The subtrajectory-clustering problem is hard for arbitrary distance functions, even from an approximation point of view, because of the infinitely large set of candidate pathlets. It is helpful to consider distance functions that are metrics, i.e., distance functions satisfying the triangle inequality, as it allows us to restrict ourselves to a finite set of candidate pathlets for computing an approximate solution of the above objective function[4]. In particular, we use the discrete Fréchet distance fr, a widely used metric for point sequences. Given two point sequences A and B, a *correspondence* is a subset $C \subseteq A \times B$ such that for all $a \in A$ (resp. $b' \in B$), there exists $b \in B$ (resp. $a' \in A$) such that $(a, b) \in C$ (resp. $(a', b') \in C$). Let $T_1 = \langle p_1, p_2, \ldots, p_k \rangle$ and $T_2 = \langle q_1, q_2, \ldots, q_\ell \rangle$ be a pair of point sequences. A correspondence C between T_1 and T_2 is monotone if for $(p_i, q_j), (p_{i'}, q_{j'}) \in C$ with $i \leqslant i'$ we have $j \leqslant j'$. The discrete Fréchet distance between T_1 and T_2 is defined as

$$\mathsf{fr}(T_1, T_2) = \min_{C \in \Xi} \max_{(p,q) \in C} ||p - q||,$$

where Ξ is the set of all monotone correspondences between $\{p_1, p_2, \ldots, p_k\}$ and $\{q_1, q_2, \ldots, q_\ell\}$.

5.3 Hardness

In this section, we show that the pathlet-cover and subtrajectory-clustering problems are both NP-hard, even when we restrict the distance function $\mathtt{d}$ to be the discrete

[4] Using a metric as a distance function makes it possible to cover every point in $\mathbb{X}$ by simply making each of the n trajectories its own pathlet. Our goal of course is to find a much smaller pathlet dictionary that pulls pathlets from the commonly traversed portions of the trajectory collection.

Fréchet distance fr. The decision version of the pathlet-cover problem can be formulated as follows: given an instance $(\mathcal{T}, \mathbb{P}, \mathsf{d})$ of pathlet-cover and a value k, determine whether $\pi(\mathcal{T}, \mathbb{P}, \mathsf{d}) \leqslant k$. We define the decision version of subtrajectory-clustering similarly.

We reduce the Facility-Location problem, known to be NP-complete [103], to the pathlet-cover problem. Given two sets of points F and C referred to as the facilities and customers, respectively, a facility cost $c > 0$, and a real number $k' > 0$, the Facility-Location problem asks if there exists a subset $F' \subseteq F$ of facilities and an assignment of customers to facilities $f : C \to F'$ such that $c|F'| + \sum_{p \in C} ||f(p) - p|| \leqslant k'$.

Consider the following reduction to the pathlet-cover problem from Facility-Location. Initially, we set $\mathcal{T}$ and $\mathbb{P}$ to be empty. For each customer $p \in C$, we add a trajectory $T_p = \langle p \rangle$ to $\mathcal{T}$. For each facility $f \in F$, we add a candidate pathlet $P_f = \langle f \rangle$ to $\mathbb{P}$. Finally, we let $c_1 = c$, $c_2 > k'$, and $c_3 = 1$, and we solve the decision version of pathlet-cover over $(\mathcal{T}, \mathbb{P}, \mathsf{fr})$ with $k = k'$. Observe $\mathsf{fr}(T_p, P_f) = ||p - f||$. We derive the following theorem[5].

Theorem 53. *The pathlet-cover problem is NP-complete.*

Similarly, we can reduce the variant of Facility-Location where F is implicitly defined as all points in $\mathbb{R}^2$ to the subtrajectory-clustering problem using essentially the same reduction[6].

Theorem 54. *The subtrajectory-clustering problem is NP-hard.*

[5] The proof uses the discrete Fréchet distance for the distance function d used to measure similarity between pathlets and subtrajectories. If we allow arbitrary distance functions instead, then we can show the stronger result that there exists no $o(\log m)$-approximation for pathlet-cover unless P = NP.

[6] While we are able to prove hardness for subtrajectory-clustering, we do not have a proof that the problem is in NP because we do not know if the number of bits required to describe optimal dictionary pathlets is polynomial in the input size.

5.4 Pathlet-cover

In this section, we describe an approximation algorithm for the pathlet-cover problem: the algorithm relies on a reduction to the standard Set-Cover problem. We define a set system as a pair $(X, \mathcal{S})$, where $X = \{e_1, \ldots, e_\ell\}$ is the ground set of elements and $\mathcal{S}$ is a family of subsets of X. Given a set system $(X, \mathcal{S})$ and a weight function $\mathrm{w} : \mathcal{S} \to \mathbb{R}^+$, the optimization version of the Set-Cover problem asks for a subset $\mathcal{C} \subseteq \mathcal{S}$ of minimum total weight such that $\bigcup \mathcal{C} = X$. Below, we describe an approximation-preserving reduction from the pathlet-cover problem to the Set-Cover problem. Running the classic greedy algorithm for Set-Cover as described below results in an $O(\log m)$-approximation for both the Set-Cover and original pathlet-cover instances. Unfortunately, the reduction constructs an exponentially large set system. Our main algorithmic challenge is to implicitly run the greedy algorithm without having to explicitly create the whole set system.

5.4.1 From pathlet-cover to set-cover

Fix an instance $(\mathcal{T}, \mathbb{P}, \mathtt{d})$ of the pathlet-cover problem. Let $\mathbb{X}$ be the set of points in $\mathcal{T}$ as defined earlier; $|\mathbb{X}| = m$. We define a family of subsets $\mathcal{S}$ to create a weighted set system $(\mathbb{X}, \mathcal{S})$. There are two types of subsets in our family; the first represents trajectory points being covered by pathlets, and the second represents uncovered trajectory points. Formally, the sets are defined as follows.

1. For every $P \in \mathbb{P}$ and for any set of input subtrajectories $\mathcal{R}$ drawn from distinct trajectories of $\mathcal{T}$ such that $\mathtt{d}(S, P) \neq \infty$ for all $S \in \mathcal{R}$, family $\mathcal{S}$ contains a set $S(P, \mathcal{R}) = \{p \in S \mid S \in \mathcal{R}\}$ with $\mathrm{w}(S(P, \mathcal{R})) = c_1 + c_3 \sum_{S \in \mathcal{R}} \mathtt{d}(S, P)$.

2. For a point $p \in \mathbb{X}$, let $T^{(p)} \in \mathcal{T}$ denote the trajectory containing p. Then for every $p \in \mathbb{X}$, family $\mathcal{S}$ contains a singleton set $\{p\}$ with $\mathrm{w}(\{p\}) = c_2/|T^{(p)}|$.

As mentioned above, the first step of this reduction constructs a set system of exponential size. The following lemma expresses the correctness of our reduction.

Lemma 55. *There exists a bijection between set covers of* $(\mathbb{X}, \mathcal{S})$ *and solutions to the pathlet-cover problem for* $(\mathcal{T}, \mathbb{P}, \mathtt{d})$ *so that cost and weight remain equal across the bijection.*

Proof. Consider a solution to the pathlet-cover problem, consisting of the pathlet dictionary $\mathcal{P}$ and assignments $\mathcal{T}(P)$, for all $P \in \mathcal{P}$. We will create a solution $\mathcal{C}$ to the Set-Cover instance $(\mathbb{X}, \mathcal{S})$. For each uncovered trajectory point p, we add the singleton set $\{p\}$ to $\mathcal{C}$. For each $P \in \mathcal{P}$, we add the set $S(P, \mathcal{T}(P))$ to $\mathcal{C}$. One may easily verify that the total weight of $\mathcal{C}$ is equal to the cost of $\mathcal{P}$.

Conversely, consider a solution $\mathcal{C}$ to the Set-Cover instance. For each set in $\mathcal{C}$ of the form $S(P, \mathcal{R})$, we add the pathlet P to $\mathcal{P}$ and assign all the subtrajectories of $\mathcal{R}$ to P. We leave any points p such that $\{p\} \in \mathcal{C}$ uncovered by our pathlet dictionary. Again, the cost of our pathlet dictionary is the same as the total weight of $\mathcal{C}$. □

5.4.2 Greedy algorithm

We would like to use the standard greedy algorithm to solve the Set-Cover instance described above. The greedy algorithm picks the set that maximizes the ratio of newly covered elements to the weight of the set. We refer to these ratios as the sets' coverage-cost ratios. The algorithm continues picking sets in this manner until every element is covered. It is well-known that the greedy algorithm has an approximation ratio of $O(\log m)$ when run on the set system $(\mathbb{X}, \mathcal{S})$.

We now describe the process in more detail for our setting. In the Set-Cover instance above, each set is either a pathlet with corresponding subtrajectory assignments or a singleton set containing a trajectory point. Choosing a set $S(P, \mathcal{R})$ of the first type results in *covering* the trajectory points within subtrajectories $\mathcal{R}$ assigned

to the pathlet P. Choosing a set $\{p\}$ of the second type is implemented as marking a trajectory point $p \in \mathbb{X}$ as *permanently uncovered*. We refer to all trajectory points covered by some pathlet or marked permanently uncovered as *processed*.

Consider the state of the greedy algorithm immediately following an iteration. Let $\mathcal{C} \subseteq \mathcal{S}$ be the family of sets chosen so far by the greedy algorithm, and let $\hat{\mathbb{X}} := \mathbb{X}(\mathcal{C}) \subseteq \mathbb{X}$ be the subset of points not processed by $\mathcal{C}$. For a subtrajectory S, let $\hat{S} = S \cap \hat{\mathbb{X}}$ be the set of unprocessed points in S.

We now consider the next iteration of the greedy algorithm. For a family of subtrajectories $\mathcal{R}$ and a pathlet P, define its *coverage-cost ratio* (with respect to $\mathcal{C}$), $\rho(P, \mathcal{R})$, as

$$\rho(P, \mathcal{R}) = \frac{\sum_{S \in \mathcal{R}} |\hat{S}|}{c_1 + \sum_{S \in \mathcal{R}} c_3 \mathtt{d}(S, P)},$$

and set

$$\mathcal{T}_P = \operatorname*{arg\,max}_{\mathcal{R}: S(P,\mathcal{R}) \in \mathcal{S}} \rho(P, \mathcal{R}); \qquad P^* = \operatorname*{arg\,max}_{P \in \mathbb{P}} \rho(P, \mathcal{T}_P).$$

Similarly, for each unprocessed point $p \in \hat{\mathbb{X}}$, we define its coverage-cost ratio as $\rho(p) = |T^{(p)}|/c_2$ and set $p^* = \arg\max_{p \in \hat{\mathbb{X}}} \rho(p)$. Note that $\rho(P, \mathcal{R})$ depends on $\hat{\mathbb{X}}$ and thus on $\mathcal{C}$, while $\rho(p)$ is independent of $\mathcal{C}$.

In the next iteration, the algorithm chooses the set $S(P^*, \mathcal{T}_{P^*})$ or $\{p^*\}$, whichever has higher coverage-cost ratio. After adding the set to $\mathcal{C}$, we update the set $\hat{\mathbb{X}}$ of unprocessed points, the values $\rho(P, \mathcal{R})$, and the sets $\mathcal{T}_P$. To implement each step of the greedy algorithm, we store all pathlets and unprocessed points in a (max) priority queue with their coverage-cost ratios as the keys. At each step, we delete newly processed points from the priority queue and update the priority queue as pathlets' keys get updated. Note that pathlets remain in the priority queue even when they are added to the dictionary so that multiple copies of the same pathlet may be added to the dictionary with distinct assignments.

The main challenge in implementing a step of the greedy algorithm efficiently is the computation of $\mathcal{T}_P$ for each pathlet P since there are an exponential number of sets $S(P, \mathcal{R})$ in $\mathcal{S}$. Notwithstanding $|\mathcal{S}| = \Omega(2^n)$, we describe below an efficient procedure for computing $\mathcal{T}_P$.

5.4.3 Computing $\mathcal{T}_P$

Let $S_P(T)$ denote the set of subtrajectories S of trajectory T where $\mathtt{d}(P, S) \neq \infty$. For a set $S(P, \mathcal{R})$, we define the set of variables $x_{S,T} \in \{0, 1\}$ for each $S \in S_P(T)$ and $T \in \mathcal{T}$ which indicate whether $S \in \mathcal{R}$. We also have $\sum_{S \in S_P(T)} x_{S,T} \leqslant 1$ for all $T \in \mathcal{T}$, i.e., at most one subtrajectory of each trajectory can be in $\mathcal{R}$.

Thus, for a fixed pathlet P, computing the set $\mathcal{T}_P$ is equivalent to solving the following optimization problem.

$$\max \frac{\sum_{T \in \mathcal{T}} \sum_{S \in S(T)} |\hat{S}| x_{S,T}}{c_1 + c_3 \sum_{T \in \mathcal{T}} \sum_{S \in S(T)} \mathtt{d}(S, P) x_{S,T}}.$$

$$\text{s.t.} \quad \sum_{S \in S(T)} x_{S,T} \leqslant 1 \quad \forall T \in \mathcal{T}.$$

$$x_{S,T} \in \{0, 1\} \quad \forall S \in S(T), T \in \mathcal{T}.$$

The maximum objective value is $\geqslant \gamma$ iff there exist feasible values of the variables x such that

$$\sum_{T \in \mathcal{T}} \sum_{S \in S(T)} \left(|\hat{S}| - c_3 \gamma \mathtt{d}(S, P) \right) x_{S,T} \geqslant c_1 \gamma. \tag{5.1}$$

Define γ^* as the maximum value of γ such that there exists a valid assignment of values to the variables x that satisfy (5.1). We have $\rho(P, \mathcal{T}_P) = \gamma^*$.

For a fixed value of γ, in order to maximize the left hand side of 5.1, for each trajectory T we should pick the subtrajectory $S \in S_P(T)$ maximizing the quantity $\left(|\hat{S}| - c_3 \gamma \mathtt{d}(S, P) \right)$ provided it is greater than 0. We do not pick any subtrajectory of T if all of them make the quantity less than 0.

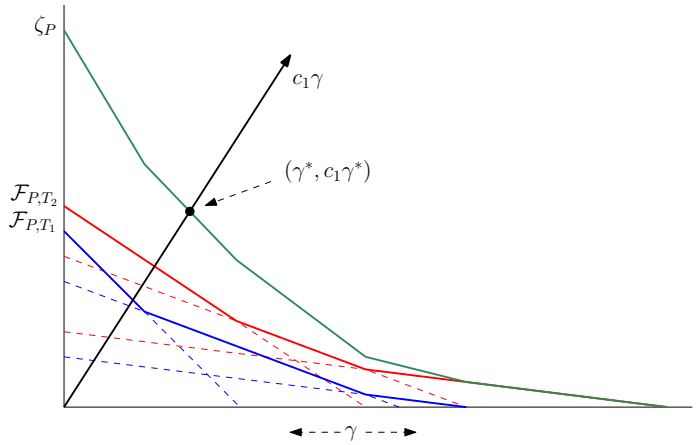


FIGURE 5.2: Functions $f_{S,P}$ for subtrajectories S of T_1 (dashed blue) and T_2 (dashed red), functions $\mathcal{F}_{P,T_1}$ (blue) and $\mathcal{F}_{P,T_2}$ (red), function ζ_P (green), and its intersection with the line of slope c_1 at γ^*.

For a subtrajectory S, let $f_{S,P}$ be a real-valued function of γ defined as $f_{S,P}(\gamma) = |\hat{S}| - c_3\gamma\mathtt{d}(S,P)$. For each pathlet P and trajectory T, we define the function $\mathcal{F}_{P,T} : \mathbb{R}_{\geqslant 0} \to \mathbb{R}_{\geqslant 0}$ as

$$\mathcal{F}_{P,T}(\gamma) = \max\{0, \max_{S \in S_P(T)} f_{S,P}(\gamma)\}.$$

Function $\mathcal{F}_{P,T}$ is a monotonically non-increasing, piecewise-linear, convex function with at most $|S_P(T)| + 1$ linear pieces. Next, we define $\zeta_P(\gamma) = \sum_{T \in \mathcal{T}} \mathcal{F}_{P,T}(\gamma)$. Function ζ_P is also a monotonically non-increasing, piecewise-linear, convex function with at most $\sum_{T \in \mathcal{T}} |S_P(T)| + m$ pieces.

Since the graph of ζ_P, which we also denote by ζ_P, is a monotonically non-increasing, convex chain, the optimal point γ^* is the intersection point of ζ_P with the line of slope c_1 and passing through the origin. See Figure 5.2. Furthermore,

$$\mathcal{T}_P = \{\arg\max_{S \in S_P(T)} f_{S,P}(\gamma^*) \mid T \in \mathcal{T}, \mathcal{F}_{P,T}(\gamma^*) > 0\}.$$

We now describe a data structure that will aid us in a binary search for γ^*.

5.4.4 Data structure

The projection of $\mathcal{F}_{P,T}$ on the γ-axis partitions it into $|S_P(T)| + 1$ intervals. Let $I_{P,T}$ denote the set of these intervals, and let $\mathcal{I}_P = \bigcup_T I_{P,T}$. We store $\mathcal{I}_P$ in a segment tree Ψ_P; see [68] for details on the segment tree. Roughly speaking, Ψ_P is a balanced binary tree. Each node v of Ψ_P is associated with an interval δ_v—the leaves are associated with "atomic" intervals between two consecutive endpoints of intervals in $\mathcal{I}_P$, and for an interval node v with children w and z, $\delta_v = \delta_w \cup \delta_z$. An interval $I \in \mathcal{I}_P$ is stored at a node v if $\delta_v \subseteq I$ and $\delta_{p(v)} \nsubseteq I$, where $p(v)$ is the parent of v. An interval is stored at $O(\log |\mathcal{I}_P|)$ nodes. Let $\mathcal{I}_{P,v} \subseteq \mathcal{I}_P$ denote the subset of intervals stored at v. We define the function $\zeta_{P,v}(\gamma) = \sum_{T:I_{P,T} \cap \mathcal{I}_{P,v} \neq \emptyset} \mathcal{F}_{P,T}(\gamma)$. The restriction of $\zeta_{P,v}$ in the interval δ_v is a linear function. We store this linear function at v. Let $\alpha(w)$ denote the set of nodes encountered on the path in Ψ_P to w. Finally, for a given leaf w, let $\zeta'_{P,w}(\gamma) = \sum_{v \in \alpha(w)} \zeta_{P,v}(\gamma)$. We have $\zeta'_{P,w}(\gamma) = \zeta_P(\gamma)$ for all $\gamma \in \delta_w$. Computing $\zeta'_{P,w}$ takes $O(\log |\mathcal{I}_P|)$ time. An interval can be inserted into or deleted from Ψ_P in $O(\log |\mathcal{I}_P|)$ time.

Returning to the greedy algorithm, whenever a new point is processed for some trajectory T, we update $\mathcal{F}_{P,T}$, the set $I_{P,T}$, and the segment tree Ψ_P. In the worst case, these updates may take $O(|S_P(T)| \log |\mathcal{I}_P|)$ time. We then perform a binary search for γ^* as mentioned earlier over the $O(|\mathcal{I}_P|)$ leaves. When searching over a leaf w of Ψ_P, we compute the intersection of $\zeta'_{P,w}$ with the line of slope c_1 passing through the origin. If the intersection occurs to the left (resp. right) of δ_w, then we continue the search over leaves to the left (resp. right) of w. Otherwise, the intersection occurs in δ_w. The binary search takes $O(\log^2 |\mathcal{I}_P|)$ time.

5.4.5 Analysis

We now analyze the running time of our algorithm. We will do so in terms of a value $\chi = \max_{P \in \mathbb{P}, T \in \mathcal{T}} |S_P(T)|$.

While this value can be as high as $O(m^2)$, we show in the next section how to reduce the value greatly when solving the subtrajectory-clustering problem for collections of κ-packed trajectories.

Selecting a pathlet or point from the priority queue takes $O(\log(b+m))$ time. Updating the pathlets in the queue at the end of each greedy step takes $O(b\log(b+m))$ time, and each point member of the queue is updated at most once in $O(\log(b+m))$ time. Therefore, the algorithm spends $O(bm\log(b+m))$ time total updating and searching the queue.

We also have $|\mathcal{I}_P| = O(n\chi)$ for any P. If a pathlet P is selected at the beginning of a greedy step, the subtrajectories it covers can be computed in $O(\log(n\chi)+k)$ time, where k is the number of newly covered subtrajectories. Therefore, finding these subtrajectories takes $O(m\log(n\chi)+m)$ time total. It takes $O(b\log^2(n\chi))$ time to run the binary searches to recompute $\rho(P,\mathcal{T}_P)$ for each candidate pathlet P at the end of a greedy step.

Finally, at the end of a greedy step, it takes $O(\chi\log\chi)$ time to recompute the function $\mathcal{F}_{P,T}$ for each pathlet P and trajectory T that has a newly processed point. There are at most m instances where a trajectory sees one or more of its points covered at the end of a greedy step, so the total time spent updating $\mathcal{F}_{P,T}$ functions is $O(bm\chi\log\chi)$. By the same argument, the total number of updates needed for each segment tree Ψ_P over all iterations is $O(m\chi)$, for a total update time of $O(bm\chi\log(n\chi))$ for all pathlets. This later bound is the bottleneck of our algorithm.

We thus have the following theorem.

Theorem 56. *Let $\mathcal{T}$ be a set of n trajectories having m points in total, $\mathbb{P}$ a set of b candidate pathlets, and* `d` *any distance function between point sequences. Let χ be the maximum number of subtrajectories of any one trajectory $T \in \mathcal{T}$ for which there are permissible assignments for any one $P \in \mathbb{P}$. Then there is an $O(\log m)$-*

approximation algorithm for the pathlet-cover problem that runs in $\tilde{O}(bm\chi)$ time, provided there is an oracle that computes the distance function `d` *in constant time.*

5.5 Subtrajectory Clustering

We now describe our approximation algorithm for the subtrajectory-clustering problem. As mentioned in the introduction, we assume trajectories to be κ-packed and the underlying distance function to be the discrete Fréchet distance. A curve is said to be κ-packed if the length of its intersection with any disk of radius r is at most κr. A point sequence is κ-packed if the polygonal curve obtained by joining its points in sequence is κ-packed.

The algorithm relies on two main ideas. First, using the fact that discrete Fréchet distance is a metric, we quickly construct a small set $\mathbb{S}$ of candidate pathlets so that the cost of the optimal pathlet cover of $\mathcal{T}$ with respect to $\mathbb{S}$ is close to that of an optimal subtrajectory clustering of $\mathcal{T}$ (Section 5.5.1)[7]. Using properties more specific to discrete Fréchet distance, we then reduce the set of candidate pathlets further to a subset $\mathbb{C}$. Next, we take advantage of our use of Fréchet distance and the input trajectories being κ-packed to quickly compute an approximation of the Fréchet distance for pathlet-cover's distance function (Section 5.5.2). This approximation enables us to consider only a small number of assignments between each remaining pathlet and trajectory, thereby greatly speeding up the algorithm without sacrificing the quality of the clustering.

5.5.1 Candidate pathlets

Let $\mathcal{T}$ be the set of n input trajectories with a total of m points, and let $\mathbb{P}$ be the set of all point sequences in $\mathbb{R}^2$. We first show how to construct a candidate set $\mathbb{S}$ of

[7] This step works for any distance function that is a metric, but for simplicity we focus on the discrete Fréchet distance.

$O(m^2)$ pathlets such that $\pi(\mathcal{T}, \mathbb{S}, \mathsf{fr}) \leqslant 2\pi(\mathcal{T}, \mathbb{P}, \mathsf{fr})$, and then show how to construct a candidate set $\mathbb{C}$ of $O(m)$ pathlets such that $\pi(\mathcal{T}, \mathbb{C}, \mathsf{fr}) = O(\log m)\pi(\mathcal{T}, \mathbb{P}, \mathsf{fr})$.

Let $\mathcal{P} \subset \mathbb{P}$ be an optimal pathlet dictionary of $\mathcal{T}$, i.e., $\mathcal{P} = \arg\min_{\mathcal{P} \subset \mathbb{P}} \mu(\mathcal{T}, \mathbb{P}, \mathsf{fr})$. Recall that for any $P \in \mathcal{P}$, $\mathcal{T}(P)$ is the family of subtrajectories assigned to P. Let $\mathtt{d}(P, \mathcal{T}(P))$ denote the cost of assigning subtrajectories in $\mathcal{T}(P)$ to P, i.e.,

$$\mathtt{d}(P, \mathcal{T}(P)) = c_3 \sum_{S \in \mathcal{T}(P)} \mathsf{fr}(S, P).$$

The lemma below states that P can be replaced by a subtrajectory of $\mathcal{T}(P)$ while only doubling the cost.

Lemma 57. *There exists a subtrajectory $S' \in \mathcal{T}(P)$ such that $\mathtt{d}(S', \mathcal{T}(P)) \leqslant 2\mathtt{d}(P, \mathcal{T}(P))$.*

Proof. Let $S' = \arg\min_{S \in \mathcal{T}(P)} \mathsf{fr}(S, P)$. We then have for any $S \in \mathcal{T}(P)$,

$$\mathsf{fr}(S, S') \leqslant \mathsf{fr}(S, P) + \mathsf{fr}(S', P) \leqslant 2\,\mathsf{fr}(S, P).$$

Thus replacing P by S' increases the cost by at most a factor of 2. □

In any solution to the pathlet-cover instance $(\mathcal{T}, \mathbb{P}, \mathsf{fr})$, we can replace all dictionary pathlets by input subtrajectories using Lemma 57, increasing the total cost by a factor of at most 2.

Corollary 58. *Let $\mathbb{P}$ be the set of all point sequences in $\mathbb{R}^2$, and let $\mathbb{S}$ be the set of all subtrajectories of $\mathcal{T}$. We have $|\mathbb{S}| = O(m^2)$ and $\pi(\mathcal{T}, \mathbb{S}, \mathsf{fr}) \leqslant 2\pi(\mathcal{T}, \mathbb{P}, \mathsf{fr})$.*

Corollary 58 immediately implies a polynomial time approximation algorithm for subtrajectory-clustering. However, we can further reduce the number of candidate pathlets by another factor of m with only an $O(\log m)$-factor increase in the approximation ratio.

Canonical pathlets. To further restrict the set of candidate pathlets beyond the set of all subtrajectories, we introduce the notion of *canonical pathlets*. Consider a trajectory $T \in \mathcal{T}$. For simplicity, we assume that $|T|$ is a power of 2. Consider a balanced binary tree over the interval $[1, |T|]$. Each node in the tree corresponds to an interval $[i, j]$, with its left and right children associated with intervals $[i, \lfloor (i+j+1)/2 \rfloor]$ and $[\lfloor (i + j + 1/2 \rfloor + 1, j]$ respectively. The leaves correspond to singleton intervals. The height of the tree is $\log_2 |T|$, and the total number of nodes is $2|T|$. Each interval induces a subtrajectory $T(i, j)$ of T. These subtrajectories of T corresponding to the intervals of the tree nodes are called the canonical pathlets of T, which we denote by $\mathbb{C}(T)$. Applying the same procedure to all the trajectories, we get a set $\mathbb{C}$ of $O(m)$ canonical pathlets.

We can replace a subtrajectory pathlet P by a set of $O(\log m)$ canonical pathlets $C(P)$ while increasing the assignment cost by at most a factor of $O(\log m)$. Recall $\mathsf{d}(P, \mathcal{T}(P)) = \sum_{S \in \mathcal{T}(P)} \mathsf{fr}(S, P)$. The lemma below formalizes the prior observation.

Lemma 59. *For a pathlet P and any set of subtrajectories $\mathcal{T}(P)$, there exists a set of $O(\log m)$ canonical pathlets $C(P) = \{P_1, \ldots, P_{|C(P)|}\}$ and a family of subtrajectory sets $\{\mathcal{T}(P_1), \ldots, \mathcal{T}(P_{|C(P)|})\}$, the union of whose points equals the points in $\mathcal{T}(P)$, such that*

$$\sum_{P_i \in C(P)} \mathsf{d}(P_i, \mathcal{T}(P_i)) = O(\log m) \cdot \mathsf{d}(P, \mathcal{T}(P)).$$

Proof. Let P be a subtrajectory of trajectory T. It is not hard to see that P can be written as the concatenation of pathlets $P_1, P_2, \ldots, P_{|C(P)|}$ in order, where each $P_i \in \mathcal{C}(T)$ and $|C(P)| = O(\log |T|)$. Consider a subtrajectory $S \in \mathcal{T}(P)$. Since there is a monotone correspondence between P and S with discrete Fréchet distance $\mathsf{fr}(P, S)$, there exist monotone correspondences between each P_i and some subtrajectory S_i of S such that $\mathsf{fr}(P_i, S_i) \leqslant \mathsf{fr}(P, S)$ and the S_i's concatenated in order cover S. We

construct the family of sets $\mathcal{T}(P_i)$ by including the subtrajectory S_i of S in $\mathcal{T}(P_i)$, for all $S \in \mathcal{T}(P)$ and $i \in \{1, \ldots, |C(P)|\}$. We then have

$$\begin{aligned}\sum_{P_i \in C(P)} \mathsf{d}(P_i, \mathcal{T}(P_i)) &= \sum_{S \in \mathcal{T}(P)} \sum_{i=1}^{|C(P)|} \mathsf{fr}(P_i, S_i) \\ &\leqslant |C(P)| \sum_{S \in \mathcal{T}(P)} \mathsf{fr}(P, S) \\ &= O(\log m) \cdot \mathsf{d}(P, \mathcal{T}(P)).\end{aligned}$$

□

As before, we can perform the above substitution for every subtrajectory pathlet in a (nearly) optimal dictionary with little cost in the approximation ratio.

Corollary 60. *Let $\mathbb{P}$ be the set of all point sequences in $\mathbb{R}^2$ and let $\mathbb{C}$ be the set of all canonical pathlets of $\mathcal{T}$. We have $\pi(\mathcal{T}, \mathbb{C}, \mathsf{fr}) = O(\log m) \cdot \pi(\mathcal{T}, \mathbb{P}, \mathsf{fr})$.*

5.5.2 Approximate distances

We use the set of canonical pathlets $\mathbb{C}$ as the set of candidate pathlets in our reduction to pathlet-cover. Rather than use the discrete Fréchet distance as our distance function directly, however, we instead compute a distance function $\mathtt{d}$ that approximates the Fréchet distances between a subset of pairs of canonical pathlets and subtrajectories, without decreasing the cluster quality by much. Other pairs are given a distance of ∞ to mark them as not being permissible assignments. Using $\mathtt{d}$ dramatically reduces the number of permissible assignments considered by the pathlet-cover approximation algorithm, and we are able to compute approximate Fréchet distances much faster than we can compute them exactly. Our algorithm for computing these distances is based upon a simple trajectory simplification scheme as described below.

For any $r > 0$, an *r-simplification* of a trajectory T is a trajectory S consisting of a subsequence of points from T such that $\mathsf{fr}(T, S) \leqslant r$. The following is a well-

known linear-time greedy algorithm to compute an r-simplification of T. We include the first point of T. We then iterate along the points of T in order, and include in our simplification each point that is at a distance of at least r from the previously included point. Finally, we include the last point of T. We denote the resulting trajectory by $\overrightarrow{T_r}$. We can similarly start from the last point of T and proceed in reverse. We denote this trajectory by $\overleftarrow{T_r}$. Finally, let T_r denote the merger of $\overrightarrow{T_r}$ and $\overleftarrow{T_r}$ obtained by including each point that appears in either trajectory in the order they originally appeared in T. The following lemma follows from the construction:

Lemma 61. (i) Let p' and q' appear in T_r in order. We have $\mathsf{fr}(T[p', q'], T_r[p', q']) \leqslant r$.

(ii) Let p and q appear in T in order. There exist $p', q' \in T_r$ such that $T[p, q]$ is a subsequence of $T[p', q']$ and $\mathsf{fr}(T[p, q], T_r[p', q']) \leqslant r$.

(iii) The lengths of the edges in $\overrightarrow{T_r}$ (resp. $\overleftarrow{T_r}$), except possibly the last (resp. first) edge, is at least r.

As described below, computing r-simplifications of trajectories allows us to quickly compute Fréchet distances within an additive error of $O(r)$. In addition, for any pair $P \in \mathbb{C}, T \in \mathcal{T}$, the number of subtrajectories of T_r, the r-simplification of T, within distance $O(r)$ of P is a constant, assuming κ is a constant.

Let σ denote the *spread* of the trajectories' point sets, i.e., the ratio between the maximum and the minimum pairwise point distances. (Recall our assumption that points are distinct; while the assumption aids in our presentation, the algorithm can be easily adapted to the case when a point appears in multiple trajectories.)

We now construct a distance function d that gives a 4-approximation for the discrete Fréchet distance fr for some pathlet-subtrajectory pairs. For others, their distance will be set so corresponding assignments are not permissible. Let $\underline{r}$ (resp. $\bar{r}$) be the minimum (resp. maximum) pairwise distance between trajectory points:

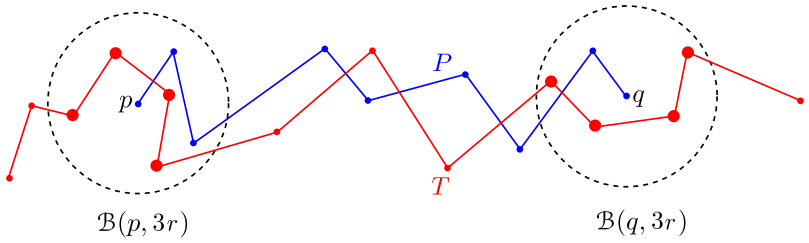


FIGURE 5.3: Sets Q^1_{TPr} and Q^2_{TPr} as the thick red points inside the left and right balls, respectively.

$\bar{r} = \sigma\underline{r}$.

For each $T \in \mathcal{T}, r \in \langle \bar{r}, \bar{r}/2, \ldots, \underline{r} \rangle$, we do the following. We compute T_r in $|T|$ time as described above. We want to efficiently decide which subtrajectories of T_r are within distance $O(r)$ of each pathlet P. To do so, we preprocess the set of points in T_r into an approximate spherical range query data structure of size $O(|T_r|)$ in $O(|T_r|)$ time; see Aronov *et al.* [33]. The data structure can answer queries of the following form: Let $\mathcal{B}(p, r) = \{x \in \mathbb{R}^2 \mid ||x - p|| \leqslant r\}$ denote a ball of radius r around p. Given a point p, the data structure returns a subset of points of $\mathcal{B}(p, 2r) \cap T_r$, including all points of $\mathcal{B}(p, r) \cap T_r$. It returns no points outside $\mathcal{B}(p, 2r) \cap T_r$. Each query takes constant time[8], plus additional time proportional to the number of points returned. For each $P \in \mathbb{C}$, let p and q be the start and end points of P, respectively. Let $Q^1_{TPr} = \mathcal{B}(p, 3r) \cap T_r$ and $Q^2_{TPr} = \mathcal{B}(q, 3r) \cap T_r$ (see Figure 5.3). We compute Q^1_{TPr} and Q^2_{TPr} using the approximate spherical range query data structure, and then for every pair (p', q') such that $p' \in Q^1_{TPr}$ and $q' \in Q^2_{TPr}$, we invoke a decision procedure that checks if $\mathsf{fr}(T_r[p', q'], P)$ is at most $3r$; see [74, Lemma 3.1]. Suppose $\mathsf{fr}(T_r[p', q'], P) \leqslant 3r$. We set $\mathtt{d}(T[p', q'], P) = 4r$. We do the above for all triples (T, r, P), and we let all other values of $\mathtt{d}(\cdot, \cdot)$ not assigned above be equal to ∞. For our algorithm, we return an $O(\log m)$-approximate pathlet dictionary for the pathlet-cover instance $(\mathcal{T}, \mathbb{C}, \mathtt{d})$ using Theorem 56.

[8] This constant is exponential in the ambient dimension.

5.5.3 Analysis

We now turn to analyzing the running time and approximation factor of our subtrajectory-clustering algorithm.

Running time. Our main goals are to bound the number of permissible assignments per pathlet-trajectory pair and the time it takes to compute their Fréchet distances. The following lemma helps with both goals.

Lemma 62. *For each $P \in \mathbb{C}$, $T \in \mathcal{T}$, and $r \in \{\bar{r}, \bar{r}/2, \ldots, \underline{r}\}$, we have $|Q^1_{TPr}|, |Q^2_{TPr}| \leqslant O(\kappa)$.*

Proof. Let p be the starting point of P. By Lemma 61(iii), each pair of consecutive points along $\overrightarrow{T_r}$ (resp. $\overleftarrow{T_r}$) are distance at least r apart (except possibly at the endpoints). In particular, there is at least r length of curve from $\overrightarrow{T_r}$ (resp. $\overleftarrow{T_r}$) lying in $\mathcal{B}(p, 4r)$ between every pair of consecutive points in $Q^1_{TPr} \cap \overrightarrow{T_r}$ (resp $Q^1_{TPr} \cap \overleftarrow{T_r}$). By the definition of κ-packed curves, there are only $O(\kappa)$ such curve portions within that ball. A similar argument holds for Q^2_{TPr}. □

As in the previous section, for each canonical pathlet $P \in \mathbb{C}$ and trajectory $T \in \mathcal{T}$, let $S_P(T)$ denote the set of subtrajectories S of trajectory T where $\mathtt{d}(P, S) \neq \infty$. Recall the value $\chi = \max_{P \in \mathbb{C}, T \in \mathcal{T}} |S_P(T)|$. The following lemma bounds χ.

Lemma 63. *For each $P \in \mathbb{C}, T \in \mathcal{T}$, we have $|S_P(T)| \leqslant O(\kappa^2 \log \sigma)$.*

Proof. By Lemma 62, for a fixed value $r \in \{\bar{r}, \bar{r}/2, \ldots, \underline{\mathrm{r}}\}$, the total number of subtrajectories for which $\mathtt{d}(T, P)$ is set to be finite is $O(|Q^1_{TPr}|.|Q^2_{TPr}|) = O(\kappa^2)$. There are $O(\log \sigma)$ values of r considered by the algorithm. □

The algorithm spends $O(m \log \sigma)$ time total simplifying trajectories. Fix a canonical pathelet P, trajectory T, and resolution r. Let p and q be the first and last

points of P. One can show $\mathcal{B}(p, 6r) \cap T_r$ and $\mathcal{B}(q, 6r) \cap T_r$ both contain $O(\kappa)$ points. Therefore, it takes $O(\kappa)$ time to compute Q^1_{TPr} and Q^2_{TPr} using the approximate spherical range-query data structure. For any subtrajectory S of T_r, testing if $\mathsf{fr}(S, P) \leqslant 3r$ can be done in $O(\kappa|P|)$ time [74]. There are $b = O(m)$ canonical pathlets, and their total length (i.e., number of points) is $O(m \log m)$. Summing over all permissible assignments between pathlets and subtrajectories, it takes $O(\kappa mn\chi \log m) = O(\kappa^3 mn \log \sigma \log m)$ time to compute approximate Fréchet distances. Finally, it takes $\tilde{O}(bm\chi) = \tilde{O}(\kappa^2 m^2 \log \sigma)$ time to run our algorithm for pathlet-cover on the instance $(\mathcal{T}, \mathbb{C}, \mathsf{d})$. Assuming κ is a small constant, we get the following lemma.[9]

Lemma 64. *The subtrajectory-clustering algorithm runs in $\tilde{O}(m^2)$ time.*

Approximation factor. We now bound the approximation factor for our subtrajectory-clustering algorithm. In the following lemma, we claim we can replace an arbitrary assignment to a canonical pathlet with a permissible assignment offering the same coverage at approximately the same cost.

Lemma 65. *Let $P \in \mathbb{C}, T \in \mathcal{T}$. For any subtrajectory $T[p, q]$, there exist $p', q' \in \mathcal{T}$ such that $T[p, q]$ is a subsequence of $T[p', q']$ and $\mathsf{d}(T[p', q'], P) \leqslant 4\,\mathsf{fr}(T[p, q], P)$.*

Proof. Let $r \in \{\bar{r}, \bar{r}/2, \ldots, \underline{r}\}$ be such that $r \leqslant \mathsf{fr}(T[p, q], P) < 2r$. By Lemma 61(ii), there exist $p', q' \in T_r$ such that $T[p, q]$ is a subsequence of $T[p', q']$ and $\mathsf{fr}(T[p, q], T_r[p', q']) \leqslant r$. By the triangle inequality,

$$\mathsf{fr}(T_r[p', q'], P) \leqslant \mathsf{fr}(T_r[p', q'], T[p, q]) + \mathsf{fr}(T[p, q], P) < 3r.$$

By definition of the discrete Fréchet distance, $p' \in Q^1_{TPr}$, and $q' \in Q^2_{TPr}$. Therefore, when the algorithm creates T_r, it successfully verifies that $\mathsf{fr}(T_r[p', q'], P) \leqslant 3r$ and

[9] The constant inside $\tilde{O}$ depends exponentially on the ambient dimension, due to a similar dependence for the query time of the approximate spherical range query data structure of [33] .

sets $\mathsf{d}(T[p', q'], P) = 4r$. We have $\mathsf{d}(T[p', q'], P) \leqslant 4\,\mathsf{fr}(T[p, q], P)$ as promised. Note that the algorithm may later set $\mathsf{d}(T[p', q'], P)$ to be a smaller value, but doing so will not invalidate the inequality. □

We also have the following lemma establishing that the function d gives an upper bound on the discrete Fréchet distance.

Lemma 66. *Let $P \in \mathbb{C}$, and let S be a subtrajectory. We have $\mathsf{fr}(S, P) \leqslant \mathsf{d}(S, P)$.*

Proof. Suppose $\mathsf{d}(S, P) \neq \infty$. Let T be the trajectory containing S. Let r be the smallest value such that $S = T_r[p', q']$ and $\mathsf{fr}(T_r[p', q'], P) \leqslant 3r$. By Lemma 61(i), $\mathsf{fr}(T[p', q'], T_r[p', q']) \leqslant r$. Therefore, by triangle inequality,

$$\mathsf{fr}(S, P) \leqslant \mathsf{fr}(T[p', q'], T_r[p', q']) + \mathsf{fr}(T_r[p', q'], P) \leqslant 4r = \mathsf{d}(S, P).$$

□

Now, let $\mathcal{P}^*$ be the optimal solution to the subtrajectory-clustering problem. By Corollary 60, there exists a pathlet dictionary $\bar{\mathcal{P}} \subseteq \mathbb{C}$ of cost at most $O(\log m)\mu(\mathcal{T}, \mathcal{P}^*, \mathsf{fr})$. By Lemma 65, we can further modify $\bar{\mathcal{P}}$ without changing the set of covered points so that every subtrajectory assignment is permissible according to d. Let $\hat{\mathcal{P}}$ be this new dictionary. We have $\mu(\mathcal{T}, \hat{\mathcal{P}}, \mathsf{d}) \leqslant O(\log m)\mu(\mathcal{T}, \mathcal{P}^*, \mathsf{fr})$. Let $\mathcal{P} \subseteq \mathbb{C}$ be the pathlet dictionary returned by the approximate pathlet-cover algorithm run on $(\mathcal{T}, \mathbb{C}, \mathsf{d})$. By Lemma 66,

$$\begin{aligned}\mu(\mathcal{T}, \mathcal{P}, \mathsf{fr}) \leqslant \mu(\mathcal{T}, \mathcal{P}, \mathsf{d}) &\leqslant O(\log m)\mu(\mathcal{T}, \hat{\mathcal{P}}, \mathsf{d}) \\ &\leqslant O(\log^2 m)\mu(\mathcal{T}, \mathcal{P}^*, \mathsf{fr}).\end{aligned}$$

We reach our main theorem for this section.

Theorem 67. *Let $\mathcal{T}$ be a set of n trajectories in $\mathbb{R}^2$ with m points in total such that each trajectory is κ-packed for some constant κ. There is an $O(\log^2 m)$-approximation algorithm for computing a subtrajectory-clustering of $\mathcal{T}$ using the discrete Fréchet distance fr as the pathlet cost distance function that runs in $\tilde{O}(m^2)$ time.*

5.6 Preprocessing

In this section, we describe two preprocessing steps that are applied before running the greedy algorithm for pathlet-cover. First, we describe a method to interpolate missing points from sparse or lossy trajectory data. Next, we describe an approach to reduce the number of candidate pathlets considered by our pathlet-cover algorithm beyond even the canonical pathlets.

Handling sparse data. A major challenge we face in clustering, particularly when dealing with GPS data, is that individual trajectories may contain long spans where the data contains no observations. As a consequence, we lose information on what route the trajectory takes between data points. A natural approach to filling in missing observations is to compute a simple interpolation such as a line segment between pairs of consecutive data points and then add additional points to the data set along that interpolation. Unfortunately, objects do not move along straight lines even in a road network, e.g., the trajectory between two consecutive observed points may contain a turn.

Instead of computing simple interpolations based only on the pairs of consecutive trajectory data points, we use the location of observed points throughout the entire collection of trajectories to produce reasonable routes for the unknown parts of individual trajectories. Intuitively, an unknown route between two consecutive data points likely passes close by to many observed data points from other trajectories. Let δ, Δ, and ϵ be parameters. We create a graph $G = (V, E)$ as follows. We begin by creating a grid over the trajectory points using cells of side-length δ. For each grid-cell v containing at least one observed trajectory point, we add v to the vertex set V. Then, we add an edge (u, v) to E for each pair of cells u and v whose centers are within distance Δ to one-another. We give each edge (u, v) a weight equal to the

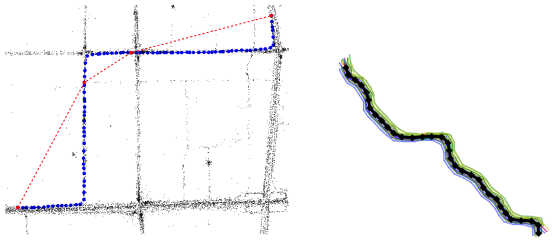


FIGURE 5.4: Left: linear (dashed red) versus our graph based (dotted blue) interpolation schemes for sparse trajectories. Right: cluster of pathlets and chosen representative pathlet in black.

inverse of the number of points lying in u and v so that edges between densely populated cells have low weight. Finally, for each pair of consecutive trajectory points p and q lying distance at least ϵ apart, we compute a shortest path in G between p and q's cells, adding the cell centers encountered along the path to the trajectory between p and q. See Figure 5.4 for a comparison of our method to the linear interpolation approach.

Sparsifying candidate pathlets. The subtrajectory clustering problem allows us to select pathlets from the full space of point sequences. By focusing just on the subtrajectories or even canonical pathlets, we are sampling from the space of candidate pathlets. Recall that the primary motivation behind our work is that large groups of subtrajectories will tend to cluster within the same proximity. Since we construct candidate pathlets from each input trajectory independently, we tend to over sample in the neighborhood of a pathlet (a point in the pathlet space) that is shared by many trajectories. As our pathlet-clustering algorithm requires time proportional to the number of candidate pathlets, we perform a sparsifying step to remove redundant canonical pathlets from overly sampled areas.

The high level idea behind our procedure is to project each canonical pathlet to Euclidean space. Similar canonical pathlets should project to closeby points. We then apply standard clustering techniques to find similar points from which only

one pathlet's point will remain a candidate. Let d and ω be parameters. For each canonical pathlet $P \in \mathbb{C}$, we select d points lying equidistant along the piecewise linear curve between P's points. Let these points be $\langle (x_1, y_1), (x_2, y_2), \ldots, (x_d, y_d) \rangle$. We create a $2d$-dimensional point $p_P = (x_1, y_1, x_2, y_2, \ldots, x_d, y_d)$. We iterate over the pathlets again, marking some as *removed* as we do so. Initially, none of the pathlets are marked. Now, consider a pathlet $P \in \mathbb{C}$ chosen during the iteration. If P is already marked as removed, we ignore it and continue to the next pathlet in the iteration. Otherwise, we consider the ball of radius ω centered at p_P and mark all other pathlets' points within the ball as removed. When we have finished iterating over the canonical pathlets, we remove the marked ones from the set of candidate pathlets used in the pathlet-cover algorithm. See Figure 5.4 for an example of a cluster of 32 similar pathlets and the one chosen in the sparsifying step.

5.7 Experiments

We describe experimental results to highlight the various desirable properties of our model mentioned earlier, and to demonstrate the efficiency of our algorithm. In particular, we show that our model can handle noisy and diverse data sets, without compromising on cluster quality. Further, our algorithm is fast, the pathlets found by our algorithm are purely data-driven (hence can be complex), and they capture the underlying shared structure. We also show the sensitivity of our results to varying parameters. All our experiments were run on an Intel Xeon 2.4 GHz computer.

Data sets. We have used both real and synthetic data sets for our experiments. The real data sets used are as follows.

(i) The Beijing data set contains taxi trajectories in Beijing, China [7]. It has month-long trajectory data of 28,000 cabs in Beijing (about 60 million points). Each trajectory originally consists of multiple trips of a taxi; we break it up so that each

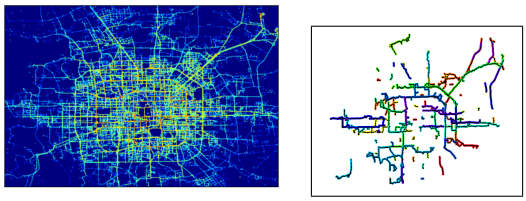


FIGURE 5.5: The heatmap (left) and major pathlets (right) of the BEIJING data set.

trip is its own trajectory. We run experiments on a subset of this data having 9 million points, spanning four days.

(ii) The GEOLIFE data set was collected by Microsoft Research Asia [6]. It tracks 182 users over four years, recording a broad range of outdoor movements, such as walking, biking, and hiking. Widely distributed in over 30 cities in China and some cities in USA and Europe, we only took trajectories in Beijing since it contained most of the data. We focus on pedestrian trajectories only, which result in 2,657 trajectories containing 1,473,115 points.

(iii) The CYCLING data set is a set of cycling trajectories in Durham, North Carolina. It consists of 37 trajectories, with 106,791 points in total. The trajectories are quite complicated, containing self-intersections and repeating portions.

Besides the above real data sets, we also used a synthetic data set, generated by the University of Minnesota Web-based Traffic Generator [8], which generates traffic data from the underlying road network in any arbitrary region of the world. The model used to generate the traffic data was given by Brinkhoff [47], which accounts for real-world conditions such as varying speeds, road congestion, etc. Denoted RTP, the data set covers the Research Triangle region [5] of North Carolina. It has around 20,000 trajectories and around 1 million points.

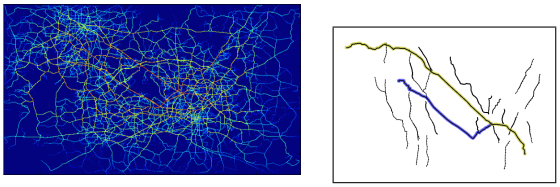


FIGURE 5.6: The heatmap (left) and major pathlets (right) of the RTP data set.

Visualization. We visually inspect the trajectories and the pathlets generated, demonstrating various desirable properties of our model and algorithm. [10]

Dense and popular portions. Our algorithm chooses pathlets from areas that have a high density of trajectory points. Figure 5.5 shows the BEIJING data set and the top 100 pathlets out of around 8,000 picked. The first pathlet chosen for BEIJING is located at the Beijing Capital International Airport (PEK), which has a high point density on the heat map. Other top pathlets include the S12 (the highway to the airport) and other highways leading into the city as well as shorter pathlets closer to the city center.

Figure 5.6 shows a portion of the RTP data set and the top 35 pathlets out of around 8,500 picked by our algorithm. The two long highlighted pathlets follow two popular highways which have a high point density on the heat map, but were chosen 26th and 32nd. Most trips within the region tend to use relatively short portions of each highway as opposed to traversing their entire length, thus other shorter pathlets were chosen earlier.

Robustness to variations. Our algorithm works well even if there is no underlying road network and data is noisy. For instance, the GEOLIFE data set is much more varied and noisy. Figure 5.8 shows the top 50 pathlets for GEOLIFE; we can see that these are in general shorter than the pathlets for BEIJING, probably because we focus on pedestrian data, and the trajectories are not as long. Figure 5.8 also shows

[10] The algorithm was run with these parameters to generate the visualizations – (c_1, c_2, c_3) values for BEIJING: $(1, 1.5, 0.005)$; GEOLIFE, CYCLING and RTP: $(1, 1, 0.005)$.

FIGURE 5.7: A bunch of trajectories from RTP (left). The blue trajectory takes a detour from the remaining trajectories, before merging again. Without gaps, the blue trajectory is picked as a pathlet on its own, with the remaining trajectories being assigned to the red pathlet (center); whereas having gaps produces three pathlets (right) with the detour being considered a gap.

a cluster of subtrajectories, which look very similar. The nice subtrajectory in the middle is picked as a pathlet, and serves as a less noisy representative for the entire cluster.

Our algorithm successfully finds common portions of trajectories that are otherwise diverse. Figure 5.10 shows trajectories from the RTP (left) and BEIJING (right) data sets passing through a common region that is captured by the highlighted pathlet.

The notion of gaps in our model gracefully handles individual variations in trajectories. We ran the algorithm on the RTP data set with a very large value of c_2 (highly penalizing gaps, and effectively removing them). See Figure 5.7.

Cluster quality. Our objective function measures cluster quality by taking the sum of distances of the subtrajectories in a cluster to the pathlet instead of other measures, such as the maximum distance to the pathlet. This results in tighter clusters (see right picture of Figure 5.9).

Trajectory reconstruction. We can approximately reconstruct individual trajectories by simply concatenating the pathlets they are assigned to (see left picture of Figure 5.9).

Data-driven pathlets. The pathlets found are completely data-driven and can be quite complex. Figure 5.11 shows a trajectory of the CYCLING data set, and

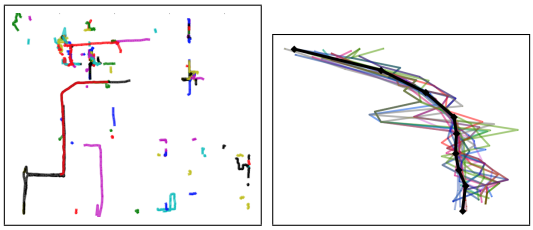


FIGURE 5.8: Major pathlets of the GEOLIFE data set (left); a cluster of subtrajectories (right) with the darker pathlet.

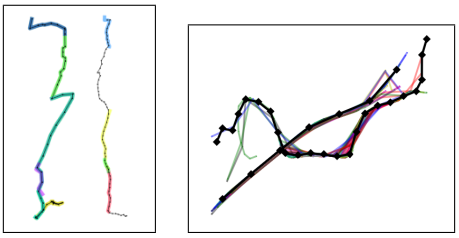


FIGURE 5.9: Left : trajectories (in black) from BEIJING and RTP resp., and the pathlets (highlighted in color) to which they are assigned. Right : a single cluster found in RTP while using the max distance to measure cluster quality, picking out only the middle darker pathlet. Using the sum of distances splits the above cluster into two – the middle pathlet's cluster, and the second pathlet's cluster.

the first two pathlets assigned to it. The trajectory contains loops, reflecting the cyclist going around a track and changing directions. Pathlets do indicate direction of travel; in particular, the first two pathlets assigned to the cycling trajectory go in each direction of the loop as shown in the figure.

Quantitative analysis. The running time of our algorithm depends upon the number of points and candidate pathlets, and complexity of the trajectories. While RTP and GEOLIFE had similar number of points, their runtimes were 1 hour and 2 hours respectively, see Table 5.1. The GEOLIFE trajectories tended to be more complex and had more intersections. The much larger BEIJING data set ran in 12 hours. It is possible to reduce runtimes by using a tighter bound on the maximum Fréchet distance, not penalizing the Fréchet distance (setting $c_3 = 0$), stopping the greedy algorithm when the increase in coverage becomes too small, and sparsifying the

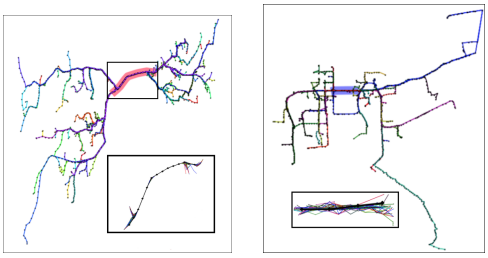


FIGURE 5.10: Full trajectories sharing a common portion, captured by pathlets with assigned subtrajectories.

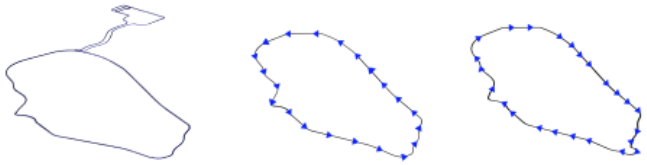


FIGURE 5.11: A self-intersecting trajectory from CYCLING (left), and the first two pathlets assigned to it.

candidate pathlets (Section 5.6).

Table 5.1 shows the effects of sparsifying the candidate pathlets. Intuitively, the sparsifying step is meant to remove redundant pathlets from the set of candidates. Accordingly, the number of candidates after sparsifying stayed roughly constant even as we took larger samples of the BEIJING data set. In general, the speed up increases as the proportion of pathlets remaining after sparsification reduces.

The pathlets picked at the beginning cover more points than those picked later. The left plot in Figure 5.12 shows the cumulative coverage (as a fraction of the total points) of pathlets picked in iterations of the greedy algorithm for the BEIJING data set, with and without sparsifiying the candidate pathlets; one can see that the plots are almost identical. Further, the marginal increase in coverage goes down drastically for later pathlets.

We also investigated the number of pathlets assigned per trajectory. The left plot in Figure 5.13 shows the frequency of trajectories vs. the number of pathlets assigned

Data Set	# points	Runtime (hours)	Runtime after sparsifying	# pathlets	# pathlets after sparsifying
RTP	1,084,257	1.02	0.54	150,142	10,217
GeoLife	1,473,115	2.05	0.83	213,478	11,456
Beijing	9,121,035	12.26	8.12	520,138	18,277
Beijing	20,012,380	50.39	21.08	1,268,598	22,651

Table 5.1: Runtimes showing speed-up with pathlet sparsification. (c_1, c_2, c_3) values: GeoLife and RTP: $(1, 1, 0.005)$, smaller Beijing: $(1, 1.5, 0.005)$, larger Beijing: $(1, 1.5, 0)$;

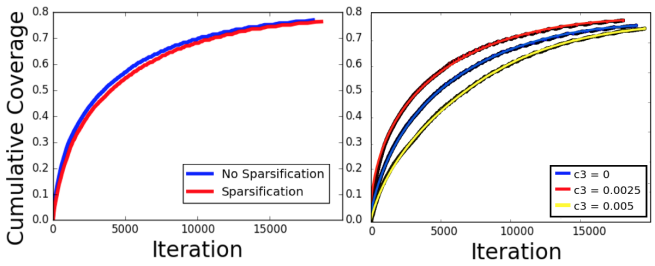


Figure 5.12: The cumulative coverage of chosen pathlets with and without sparsification (left). Cumulative coverage plots for different values of c_3 (with $c_1 = 1, c_2 = 1.75$) (right). Both plots are for Beijing.

per trajectory for the Beijing data set. There is a very long tail, meaning that there exist trajectories with a lot of pathlets assigned to them. However, the tail mostly consists of pathlets added later by the algorithm; these are shorter and many of these are assigned to a single trajectory on average. The right plot in Figure 5.13 shows the same plot for the top 50 and 100 pathlets. The top pathlets usually concatenate in smaller numbers.

Parameter sensitivity. We conclude by considering the effect each parameter has on the pathlets chosen as well as the coverage of the trajectories' points.

Keeping c_1 and c_3 fixed, we vary c_2. A lower value of c_2 means it is less expensive

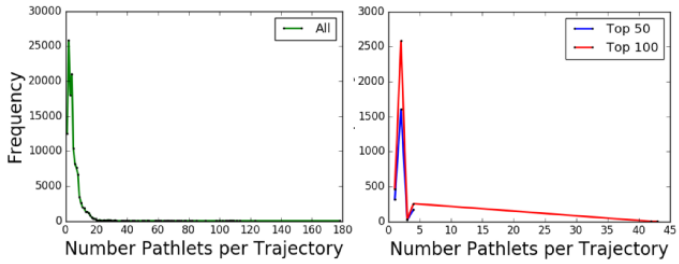


FIGURE 5.13: Plots showing the number of pathlets assigned per trajectory.

to leave points unassigned to any pathlet; thus increasing c_2 increases the percentage of assigned points, e.g., from around 64% ($c_2 = 1.5$) to 84% ($c_2 = 2.5$) for BEIJING. However, the data itself also affects how much of its points get assigned. To get the same percentage of assigned points of around 74%, we need $c_2 = 2$ for BEIJING and $c_2 = 1$ for RTP and GEOLIFE. Urban centers lead to more route diversity (and somewhat lesser shared structure), hence we need larger c_2 values for BEIJING to get the same percentage of assigned points, whereas traffic tends to cluster along the relatively few long highways in RTP. For GEOLIFE, although the c_2 values are similar to that of RTP, in general the number of pathlets picked is less than RTP (perhaps because in RTP there are more shared portions, and there are a lot less trajectories and hence more points per trajectory). The average number of subtrajectories assigned per pathlet reduces as c_2 increases. Thus the pathlets cover less shared structure.

We now vary c_3 while keeping c_1 and c_2 fixed. The right plot in Figure 5.12 shows the impact on the cumulative coverage in BEIJING as pathlets are added for different values of c_3. The number of pathlets chosen increases with c_3, from $8,169$ for $c_3 = 0$, to $10,021$ for $c_3 = 0.0025$, to $11,223$ for $c_3 = 0.005$. Similar trends are observed for other data sets. As our experiments suggest, increasing c_3 discourages the assignment of many long subtrajectories to individual pathlets as the distance between these subtrajectories and pathlets has a larger and larger influence on our

objective function.

5.8 Conclusion

We presented a model for constructing a pathlet dictionary and considered two optimization problems based on our model, pathlet-cover and subtrajectory clustering. After presenting hardness results, we described polynomial time polylogarithmic approximation algorithms for both problems. Finally, we evaluated our model and algorithms through experiments, and visualization and quantitative analysis of the results.

It would be interesting to see how our techniques could be generalized or applied to other settings. In particular, we would like to know if our model can be applied in contexts such as motion capture or if our techniques could be used to fit generative models to trajectory data. It would also be good to know if our algorithms could be implemented in parallel or I/O efficiently without sacrificing much in solution quality.

6

Conclusion

We conclude by briefly summarizing the contributions of this dissertation, and also highlighting some important general research directions.

Parallel algorithms. We designed algorithms to compute, store and query distributed data structures for answering orthogonal range searching and nearest-neighbor queries in the MPC model of computation. We also proposed an MPC algorithm to compute the Delaunay triangulation of a point set in 2D, and the contour tree of a real-valued, piecewise linear function defined on the triangulation. Our algorithms are the first ones to have theoretical guarantees on their performance, and are asymptotically optimal or near-optimal.

It would be interesting to see if such provably-efficient algorithms can also be designed for other geometric and topological problems, e.g., point location queries, persistent homology etc. Several basic problems are still open in the MPC model, e.g., graph connectivity in sub-logarithmic rounds. See also [102] for further open problems in parallel query processing in the MPC model.

Gromov-Hausdorff distance. We prove the first hardness results for computing the Gromov-Hausdorff distance between two metric spaces, showing that it is NP-hard to approximate to a factor better than 3, even for tree metrics. We also give a $O(\min\{n, \sqrt{rn}\})$-factor approximation algorithm for tree metrics, where r is the ratio of the length of the longest to the shortest edge. We also give algorithms to compute the interleaving distance between merge trees with similar performance guarantees.

An important open problem is to close the gap between the upper and lower bounds on the approximation factor. However, this seems challenging since the key technique of using a graph isomorphism does not seem to lead to a better approximation. We hope our work further stimulates research in the area of computing correspondences between metric spaces having low *additive* distortion.

Subtrajectory clustering. We propose a new model for clustering subtrajectories. Such clusters capture the shared movement patterns in input trajectories in the form of a representative *pathlet* for each cluster. We prove hardness results for the subtrajectory clustering problem. We also give approximation algorithms for the same, which can be made faster for realistic input trajectories. We also demonstrate the practicality of our algorithm by performing extensive experiments.

While our algorithms are provably efficient, we use a heuristic approach to further speed up our experiments – we reduce the number of candidate pathlets by running a single-linkage type procedure to cluster the set of candidate pathlets. This gives rise to an important question concerning the clustering of curves (under say the discrete Fréchet distance) for optimizing the k-median, k-means, and k-center objectives. Apart from results that apply to general metric spaces, very little is known about algorithms specific to clustering curves with provable performance guarantees. The closest work is the one by Driemel *et al.* [75] on clustering time-series data (i.e., sequences of real numbers); however their work does not seem to extend easily to

curves. One important hurdle is the fact that the Fréchet distance has infinite doubling dimension, so the sampling framework of [11] does not work. Another approach is to look beyond worst-case instances, taking a cue from the "CDNM" thesis – Clustering is Difficult only when it does Not Matter [66]. Various notions of *clusterability* of data sets have been developed [10], and many clustering problems that are hard to solve in the worst case can be solved optimally for such instances [29, 37, 143].

Bibliography

[1] https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.

[2] https://www.nature.com/news/the-fight-to-save-thousands-of-lives-with-sea-floor-sensors-1.22178.

[3] https://cacm.acm.org/magazines/2017/9/220427-data-sketching/fulltext.

[4] http://pig.apache.org/.

[5] Research Triangle, 2003.

[6] Geolife. http://research.microsoft.com/en-us/projects/GeoLife/, 2009.

[7] Taxi trajectory open dataset, Tsinghua University, China, 2009. http://sensor.ee.tsinghua.edu.cn.

[8] MNTG: Minnesota web-based traffic generator, 2013. http://mntg.cs.umn.edu/tg/index.php.

[9] A. Acharya and V. Natarajan. A parallel and memory efficient algorithm for constructing the contour tree. In *Proc. IEEE Pacific Vis. Symp.*, pages 271–278, 2015.

[10] M. Ackerman and S. Ben-David. Clusterability: A theoretical study. In *Artif. Intell. Stat.*, pages 1–8, 2009.

[11] M. R. Ackermann, J. Blömer, and C. Sohler. Clustering for metric and non-metric distance measures. *ACM Trans. Alg.*, 6(4):59, 2010.

[12] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a MapReduce computation. *Proc. VLDB Endow.*, 6(4):277–288, 2013.

[13] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. Int. Colloq. Automata, Lang. Program.*, pages 115–127, 2001.

[14] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient construction of constrained Delaunay triangulations. In *Proc. Euro. Symp. Algo.*, pages 355–366, 2005.

[15] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient batched union-find and its applications to terrain analysis. *ACM Trans. Alg.*, 7(1):11:1–11:21, 2010.

[16] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. J. Haverkort. Box-trees and R-trees with near-optimal query time. *Disc. Computat. Geom.*, 28(3):291–312, 2002.

[17] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. Goodman, and R. Pollack, editors, *Adv. Disc. Computat. Geom.*, pages 1–56. American Mathematical Society, 1998.

[18] P. K. Agarwal, K. Fox, K. Munagala, and A. Nath. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *Proc. ACM Symp. Princ. Database Sys.*, pages 429–440. ACM, 2016.

[19] P. K. Agarwal, K. Fox, K. Munagala, A. Nath, J. Pan, and E. Taylor. Subtrajectory clustering: Models and algorithms. In *Proc. ACM Symp. Princ. Database Sys.*, pages 75–87. ACM, 2018.

[20] P. K. Agarwal, K. Fox, and A. Nath. Maintaining reeb graphs of triangulated 2-manifolds. In *LIPIcs-Leibniz Int. Proc. Informatics*, volume 93. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[21] P. K. Agarwal, K. Fox, A. Nath, A. Sidiropoulos, and Y. Wang. Computing the Gromov-Hausdorff distance for metric trees. *ACM Trans. Alg.*, 14(2):24, 2018.

[22] P. K. Agarwal, T. Mølhave, M. Revsbæk, I. Safa, Y. Wang, and J. Yang. Maintaining contour trees of dynamic terrains. In *Proc. Int. Symp. Computat. Geom.*, pages 796–811, 2015.

[23] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, 1988.

[24] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[25] I. Akhter, Y. Sheikh, S. Khan, and T. Kanade. Trajectory space: A dual representation for nonrigid structure from motion. *IEEE Trans. Pattern Analy. Mach. Intell.*, 33(7):1442–1456, 2011.

[26] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow model: A practical approach to balancing correctness, latency,

and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

[27] L. Alarabi, A. Eldawy, R. Alghamdi, and M. F. Mokbel. TAREEG: a MapReduce-based system for extracting spatial data from OpenStreetMap. In *Proc. ACM Int. Conf. Advances Geog. Inf. Sys.*, pages 83–92, 2014.

[28] A. Andoni, A. Nikolov, K. Onak, and G. Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proc. ACM Symp. Theory Comput.*, pages 574–583, 2014.

[29] H. Angelidakis, K. Makarychev, and Y. Makarychev. Algorithms for stable and perturbation-resilient problems. In *Proc. ACM Symp. Theory Comput.*, pages 438–451. ACM, 2017.

[30] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. Resende, editors, *Handbook of Massive Data Sets*, pages 313–357. Springer, 2002.

[31] L. Arge, G. S. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*. CRC Press, 2005.

[32] L. Arge, M. Revsbæk, and N. Zeh. I/O-efficient computation of water flow across a terrain. In *Proc. ACM Symp. Computat. Geom.*, pages 403–412, 2010.

[33] B. Aronov, S. Har-Peled, C. Knauer, Y. Wang, and C. Wenk. Fréchet distance for curves, revisited. In *Proc. Euro. Symp. Alg.*, pages 52–63, 2006.

[34] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[35] F. Aurenhammer, R. Klein, and D.-T. Lee. *Voronoi Diagrams and Delaunay Triangulations*. World Scientific, 2013.

[36] B. Bahmani, A. Goel, and R. Shinde. Efficient distributed locality sensitive hashing. In *Proc. ACM Int. Conf. Info. Know. Manage.*, pages 2174–2178, 2012.

[37] M.-F. Balcan, A. Blum, and A. Gupta. Approximate clustering without the approximation. In *Proc. ACM-SIAM Symp. Disc. Alg.*, pages 1068–1077. Society for Industrial and Applied Mathematics, 2009.

[38] U. Bauer, X. Ge, and Y. Wang. Measuring distance between Reeb graphs. In *Proc. Symp. Computat. Geom.*, pages 464–473, 2014.

[39] U. Bauer, M. Kerber, and J. Reininghaus. Clear and compress: Computing persistent homology in chunks. In *Topological Methods in Data Analysis and Vis. III*, pages 103–117. Springer, 2014.

[40] U. Bauer, M. Kerber, and J. Reininghaus. Distributed computation of persistent homology. In *Proc. Workshop Alg. Engg. Exp.*, pages 31–38. SIAM, 2014.

[41] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *Proc. ACM Symp. Princ. Database Sys.*, pages 273–284, 2013.

[42] K. G. Bemis, D. Silver, P. A. Rona, and C. Feng. Case study: a methodology for plume visualization with application to real-time acquisition and navigation. In *Proc. IEEE Conf. Vis.*, pages 481–484, 2000.

[43] H. B. Bjerkevik and M. B. Botnan. Computational complexity of the interleaving distance. In *Proc. Int. Symp. Computat. Geom.*, pages 13:1–13:15, 2018.

[44] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Research*, 3:993–1022, 2003.

[45] G. E. Blelloch, J. C. Hardwick, G. L. Miller, and D. Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24(3-4):243–269, 1999.

[46] P. Bose and L. Devroye. On the stabbing number of a random Delaunay triangulation. *Computat. Geom.*, 36(2):89–105, 2007.

[47] T. Brinkhoff. Generating network-based moving objects. In *Proc. Int. Conf. Scien. Statis. Database Manag.*, pages 253–255. IEEE, 2000.

[48] A. M. Bronstein, M. M. Bronstein, and R. Kimmel. Efficient computation of isometry-invariant distances between surfaces. *SIAM J. Sci. Comp.*, 28(5):1812–1836, 2006.

[49] K. Buchin, M. Buchin, D. Duran, B. T. Fasy, R. Jacobs, V. Sacristan, R. I. Silvera, F. Staals, and C. Wenk. Clustering trajectories for map construction. In *Proc. ACM Int. Conf. Advances Geog. Info. Sys.* ACM, 2017.

[50] K. Buchin, M. Buchin, J. Gudmundsson, M. Löffler, and J. Luo. Detecting commuting patterns by clustering subtrajectories. *Int. J. Computat. Geom. & Appl.*, 21(03):253–282, 2011.

[51] K. Buchin, M. Buchin, M. Van Kreveld, M. Löffler, R. I. Silveira, C. Wenk, and L. Wiratma. Median trajectories. *Algorithmica*, 66(3):595–614, 2013.

[52] K. Buchin, M. Buchin, M. Van Kreveld, and J. Luo. Finding long and similar parts of trajectories. *Computat. Geom.*, 44(9):465–476, 2011.

[53] M. Buchin, A. Driemel, M. van Kreveld, and V. Sacristán. Segmenting trajectories: A framework and algorithms using spatiotemporal criteria. *J. Spat. Inf. Sc.*, (3):33–63, 2014.

[54] D. Burago, Y. Burago, and S. Ivanov. *A Course in Metric Geometry.* American Mathematical Society, 2001.

[55] G. Carlsson and F. Mémoli. Characterization, stability and convergence of hierarchical clustering methods. *J. Mach. Learn. Research*, 11:1425–1470, 2010.

[56] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computat. Geom.*, 24(2):75–94, 2003.

[57] H. Carroll, M. J. Clement, P. Ridge, and Q. O. Snell. Effects of gap open and gap extension penalties. 2006.

[58] T. M. Chan. Optimal partition trees. *Disc. Computat. Geom.*, 47(4):661–690, 2012.

[59] F. Chazal, D. Cohen-Steiner, L. J. Guibas, F. Mémoli, and S. Y. Oudot. Gromov-Hausdorff stable signatures for shapes using persistence. In *Computer Graphics Forum*, volume 28, pages 1393–1403. Wiley Online Library, 2009.

[60] B. Chazelle. *The Discrepancy Method - Randomness and Complexity.* Cambridge University Press, 2001.

[61] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3):229–249, 1990.

[62] C. Chen, H. Su, Q. Huang, L. Zhang, and L. Guibas. Pathlet learning for compressing and planning trajectories. In *Proc. ACM Int. Conf. Adv. Geog. Inf. Sys.*, pages 382–385. ACM, 2013.

[63] L. P. Chew, M. T. Goodrich, D. P. Huttenlocher, K. Kedem, J. M. Kleinberg, and D. Kravets. Geometric pattern matching under Euclidean motion. *Computat. Geom.*, 7(1-2):113–124, 1997.

[64] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009.

[65] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. A. Ramos. Randomized external-memory algorithms for line segment intersection and other geometric problems. *Int. J. Computat. Geom. Appl.*, 11(3):305–337, 2001.

[66] A. Daniely, N. Linial, and M. Saks. Clustering is difficult only when it does not matter. *arXiv preprint arXiv:1205.4891*, 2012.

[67] A. Danner, T. Mølhave, K. Yi, P. K. Agarwal, L. Arge, and H. Mitásová. TerraStream: from elevation data to watershed hierarchies. In *Proc. Int. Symp. Geog. Inf. Sys.*, page 28, 2007.

[68] M. de Berg, M. van Kreveld, M. Overmars, and O. Cheong. *Computational Geometry*. Springer, 3rd edition, 2000.

[69] V. De Silva, E. Munch, and A. Patel. Categorified Reeb graphs. *Disc. Computat. Geom.*, 55(4):854–906, 2016.

[70] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Comm. ACM*, 51(1):107–113, 2008.

[71] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. Symp. Computat. Geom.*, pages 298–307, 1993.

[72] T. K. Dey, D. Shi, and Y. Wang. Comparing graphs via persistence distortion. *CoRR*, abs/1503.07414, 2015.

[73] T. K. Dey, D. Shi, and Y. Wang. Comparing graphs via persistence distortion. In *Proc. Int. Symp. Computat. Geom.*, pages 491–506, 2015.

[74] A. Driemel, S. Har-Peled, and C. Wenk. Approximating the Fréchet distance for realistic curves in near linear time. *Disc. Computat. Geom.*, 48(1):94–127, 2012.

[75] A. Driemel, A. Krivošija, and C. Sohler. Clustering time series under the Fréchet distance. In *Proc. ACM-SIAM Symp. Disc. Alg.*, pages 766–785. Society for Industrial and Applied Mathematics, 2016.

[76] M. Elad and M. Aharon. Image denoising via sparse and redundant representations over learned dictionaries. *IEEE Trans. Image Process.*, 15(12):3736–3745, 2006.

[77] A. Eldawy, L. Alarabi, and M. F. Mokbel. Spatial partitioning techniques in SpatialHadoop. *Proc. VLDB Endow.*, 8(12):1602–1613, 2015.

[78] A. Eldawy, Y. Li, M. F. Mokbel, and R. Janardan. CG_Hadoop: computational geometry in MapReduce. In *Proc. ACM Int. Conf. Advances Geog. Inf. Sys.*, pages 284–293, 2013.

[79] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *Proc. IEEE Int. Conf. Data Eng.*, pages 1352–1363, 2015.

[80] A. Ene, S. Im, and B. Moseley. Fast clustering using MapReduce. In *Proc. ACM Int. Conf. Know. Discovery Data Mining*, pages 681–689, 2011.

[81] G. Evangelidis, D. B. Lomet, and B. Salzberg. The hB-Pi-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB J.*, 6(1):1–25, 1997.

[82] Q. Fan, D. Zhang, H. Wu, and K.-L. Tan. A general and parallel platform for mining co-movement patterns over large-scale trajectories. *Proc. VLDB Endow.*, 10(4):313–324, 2016.

[83] P. W. Finn, L. E. Kavraki, J.-C. Latombe, R. Motwani, C. Shelton, S. Venkatasubramanian, and A. Yao. Rapid: randomized pharmacophore identification for drug design. In *Proc. Symp. Computat. Geom.*, pages 324–333. ACM, 1997.

[84] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symp. Found. Comp. Sci.*, pages 285–298, 1999.

[85] S. Gaffney and P. Smyth. Trajectory clustering with mixtures of regression models. In *Proc. ACM Int. Conf. Know. Discovery Data Mining*, pages 63–72. ACM, 1999.

[86] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[87] A. Goel and K. Munagala. Complexity measures for MapReduce, and comparison to parallel computing. *CoRR*, abs/1211.6526, 2012.

[88] M. T. Goodrich. Randomized fully-scalable BSP techniques for multi-searching and convex hull construction (preliminary version). In *Proc. 8th ACM-SIAM Annu. Symp. Disc. Alg.*, pages 767–776, 1997.

[89] M. T. Goodrich. Communication-efficient parallel sorting. *SIAM J. Comput.*, 29(2):416–432, 1999.

[90] M. T. Goodrich. Parallel algorithms in geometry. In *Handbook of Discrete and Computational Geometry, Second Edition.*, pages 953–967. 2004.

[91] M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching, and simulation in the MapReduce framework. In *Proc. Int. Symp. Alg. Comput.*, pages 374–383, 2011.

[92] M. Gromov. *Metric Structures for Riemannian and Non-Riemannian Spaces*. Birkhäuser Basel, 2007.

[93] J. Gudmundsson, A. Thom, and J. Vahrenhold. Of motifs and goals: mining trajectory data. In *Proc. ACM Int. Conf. Adv. Geog. Inf. Sys.*, pages 129–138. ACM, 2012.

[94] A. Hall and C. Papadimitriou. Approximating the distortion. In *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pages 111–122. Springer, 2005.

[95] D. Haussler and E. Welzl. ε-nets and simplex range queries. *Disc. Computat. Geom.*, 2:127–151, 1987.

[96] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comp.*, 2(4):225–231, 1973.

[97] C.-C. Hung, W.-C. Peng, and W.-C. Lee. Clustering and aggregating clues of trajectories for mining trajectory patterns and routes. *VLDB J.*, 24(2):169–192, 2015.

[98] D. P. Huttenlocher, K. Kedem, and J. M. Kleinberg. On dynamic voronoi diagrams and the minimum hausdorff distance for point sets under euclidean motion in the plane. In *Proc. Symp. Computat. Geom.*, pages 110–119. ACM, 1992.

[99] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *Proc. VLDB Endow.*, 4(6):385–396, 2011.

[100] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proc. ACM-SIAM Symp. Disc. Alg.*, pages 938–948, 2010.

[101] C. Kenyon, Y. Rabani, and A. Sinclair. Low distortion maps between point sets. *SIAM J. Comp.*, 39(4):1617–1636, 2009.

[102] P. Koutris, S. Salihoglu, D. Suciu, et al. Algorithmic aspects of parallel data processing. *Foundations and Trends in Databases*, 8(4):239–370, 2018.

[103] J. Krarup and P. M. Pruzan. The simple plant location problem: survey and synthesis. *Euro. J. Operational Research*, 12(1):36–81, 1983.

[104] R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani. Fast greedy algorithms in MapReduce and streaming. *ACM Trans. Parallel Comput.*, 2(3):14:1–14:22, 2015.

[105] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: A method for solving graph problems in MapReduce. In *Proc. ACM Symp. Parallel Alg. Arch.*, pages 85–94, 2011.

[106] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *Proc. ACM Int. Conf. Manage. Data*, pages 593–604. ACM, 2007.

[107] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with MapReduce: a survey. *SIGMOD Rec.*, 40(4):11–20, 2012.

[108] R. Lewis and D. Morozov. Parallel computation of persistent homology using the blowup complex. In *Proc. ACM Symp. Parallel Alg. Arch.*, pages 323–331. ACM, 2015.

[109] Y. Li, Q. Huang, M. Kerber, L. Zhang, and L. Guibas. Large-scale joint map matching of GPS traces. In *Proc. ACM Int. Conf. Adv. Geog. Info. Sys.*, pages 214–223. ACM, 2013.

[110] Y. Li, Y. Li, D. Gunopulos, and L. Guibas. Knowledge-based trajectory completion from sparse GPS samples. In *Proc. ACM Int. Conf. Adv. Geog. Info. Sys.*, page 33. ACM, 2016.

[111] Y. Li, P. M. Long, and A. Srinivasan. Improved bounds on the sample complexity of learning. *J. Comp. Sys. Sci.*, 62(3):516–527, 2001.

[112] Q. Liu and G. van Ryzin. On the choice-based linear programming model for network revenue management. *Manufac. Serv. Op. Manage.*, 10(2):233–310, 2008.

[113] D. B. Lomet and B. Salzberg. The hb-tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Sys.*, 15(4):625–658, 1990.

[114] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. ACM Int. Conf. Manage. Data*, pages 135–146, 2010.

[115] J. Matoušek. Reporting points in halfspaces. *Computat. Geom. Theory Appl.*, 2(3):169–186, 1992.

[116] J. Matoušek. Approximations and optimal geometric divide-and-conquer. *J. Comp. Sys. Sci.*, 50(2):203–208, 1995.

[117] N. Max, R. Crawfis, and D. Williams. Visualization for climate modeling. *IEEE Comp. Graphics Appl.*, 13(4):34–40, 1993.

[118] F. Mémoli. On the use of Gromov-Hausdorff distances for shape comparison. In *Proc. Symp. Point Based Graphics*, pages 81–90, 2007.

[119] F. Mémoli and G. Sapiro. A theoretical and computational framework for isometry invariant recognition of point cloud data. *Found. Computat. Math.*, 5(3):313–347, 2005.

[120] M. Mitzenmacher and E. Upfal. *Probability and Computing - Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[121] D. Morozov, K. Beketayev, and G. Weber. Interleaving distance between merge trees. *Disc. Computat. Geom.*, 49(22-45):52, 2013.

[122] D. Morozov and G. H. Weber. Distributed merge trees. In *Proc. ACM Symp. Princ. Practice Parallel Prog.*, pages 93–102, 2013.

[123] D. Morozov and G. H. Weber. Distributed contour trees. In *Topological Methods in Data Analysis and Vis. III*, pages 89–102. 2014.

[124] D. M. Mount, N. S. Netanyahu, and J. Le Moigne. Improved algorithms for robust point pattern matching and applications to image registration. In *Proc. Symp. Computat. Geom.*, pages 155–164. ACM, 1998.

[125] A. Nath, K. Fox, P. K. Agarwal, and K. Munagala. Massively parallel algorithms for computing tin dems and contour trees for large terrains. In *Proc. ACM Int. Conf. Adv. Geog. Inf. Sys.*, page 25. ACM, 2016.

[126] R. Norel, D. Fischer, H. J. Wolfson, and R. Nussinov. Molecular surface recognition by a computer vision-based technique. *Protein Engineering, Design and Selection*, 7(1):39–46, 1994.

[127] C. Notredame. Recent evolutions of multiple sequence alignment algorithms. *PLoS Computat. Bio.*, 3(8):e123, 2007.

[128] C. Panagiotakis, N. Pelekis, I. Kopanakis, E. Ramasso, and Y. Theodoridis. Segmentation and sampling of moving object trajectories based on representativeness. *IEEE Trans. Know. Data Engg.*, 24(7):1328–1343, 2012.

[129] C. Papadimitriou and S. Safra. The complexity of low-distortion embeddings between point sets. In *Proc. ACM-SIAM Symp. Disc. Algo.*, pages 112–118, 2005.

[130] N. Pelekis, P. Tampakis, M. Vodas, C. Doulkeridis, and Y. Theodoridis. On temporal-constrained sub-trajectory cluster analysis. *Data Mining and Know. Disc.*, pages 1–37, 2017.

[131] N. Pelekis, P. Tampakis, M. Vodas, C. Panagiotakis, and Y. Theodoridis. In-DBMS sampling-based sub-trajectory clustering. In *Proc. Int. Conf. Extending Database Tech.*, pages 632–643, 2017.

[132] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin. Scalable big graph processing in MapReduce. In *Proc. ACM Int. Conf. Manage. Data*, pages 827–838, 2014.

[133] J. O. Ramsay. *Functional data analysis*. Wiley Online Library, 2006.

[134] S. Sankararaman, P. K. Agarwal, T. Mølhave, J. Pan, and A. P. Boedihardjo. Model-driven matching and segmentation of trajectories. In *Proc. ACM Int. Conf. Adv. Geog. Inf. Sys.*, pages 234–243. ACM, 2013.

[135] F. Schmiedl. Computational aspects of the Gromov-Hausdorff distance and its application in non-rigid shape matching. *Disc. Computat. Geom.*, 57(4):854–880, 2017.

[136] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proc. IEEE Symp. Mass Storage Sys. Tech.*, pages 1–10, 2010.

[137] C. Sung, D. Feldman, and D. Rus. Trajectory clustering for motion prediction. In *Proc. IEEE/RSJ Int. Conf. Intell. Robots Sys.*, pages 1547–1552. IEEE, 2012.

[138] L.-A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, W.-C. Peng, and T. L. Porta. A framework of traveling companion discovery on trajectory data streams. *ACM Trans. Intell. Sys. Tech.*, 5(1):3, 2013.

[139] S. P. Tarasov and M. N. Vyalyi. Construction of contour trees in 3D in $O(n \log n)$ steps. In *Proc. Symp. Computat. Geom.*, pages 68–75, 1998.

[140] E. F. Touli and Y. Wang. FPT-algorithms for computing Gromov-Hausdorff and interleaving distances between trees.

[141] L. G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, 1990.

[142] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. Symp. Computat. Geom.*, pages 212–220, 1997.

[143] A. Vijayaraghavan, A. Dutta, and A. Wang. Clustering stable instances of Euclidean k-means. In *Proc. Adv. Neural Inf. Processing Sys.*, pages 6503–6512, 2017.

[144] S. Wang, L. Wu, F. Zhou, C. Zheng, and H. Wang. Group pattern mining algorithm of moving objects' uncertain trajectories. *Int. J. Comp. Comm. Control*, pages 428–440, 2015.

[145] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[146] H. Xu, Y. Zhou, W. Lin, and H. Zha. Unsupervised trajectory clustering via adaptive multi-kernel-based shrinkage. In *Proc. IEEE Int. Conf. Comp. Vis.*, pages 4328–4336, 2015.

[147] R. Ying, J. Pan, K. Fox, and P. K. Agarwal. A simple efficient approximation algorithm for dynamic time warping. In *Proc. ACM Int. Conf. Adv. Geog. Info. Sys.*, page 21. ACM, 2016.

[148] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. USENIX Conf. Hot Topics Cloud Comput.*, pages 10–10, 2010.

[149] K. Zheng, Y. Zheng, N. J. Yuan, and S. Shang. On discovery of gathering patterns from trajectories. In *Proc. Int. Conf. Data Engg.*, pages 242–253. IEEE, 2013.

[150] Y. Zhou and T. S. Huang. 'Bag of segments' for motion trajectory analysis. In *Proc. Int. Conf. Image Process.*, pages 757–760. IEEE, 2008.

Biography

Abhinandan Nath was born on July 3, 1991 in the city of Guwahati in the state of Assam, India. He finished most of his early education in Guwahati, and spent the last two years of his high school in Delhi Public School, R.K. Puram, New Delhi. He obtained a bachelor's degree in Computer Science and Engineering from the Indian Institute of Technology, Guwahati in 2013. He joined the PhD program in the Department of Computer Science at Duke University in the fall of 2013, and he defended his PhD dissertation on July 12, 2018.

His research interests lie in computational geometry and its applications to areas such as databases, data mining, and GIS applications. He has published and presented his work in the proceedings of peer-reviewed conferences and journals [21, 18, 125, 19, 20]. He has also reviewed papers for PODS, SIGMOD, VLDB, ICALP and ACM-GIS.

He will be joining Mentor Graphics in Fremont, California after graduation, where he will work on designing and implementing computational geometry and machine learning algorithms in Electronic Design Automation.