Trajectory Clustering using a Variation of Fréchet Distance

by

Vafa Khoshaein

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the MCS degree in
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
Faculty of Graduate and Postdoctoral Affairs
University of Ottawa

Abstract

Location-aware devices are one of the examples of variety of systems that can provide trajectory data. The formal definition of a trajectory is the path of a moving object in space as a function of time. Surveillance systems can now automatically detect moving objects and provide a useful dataset for further analysis. Clustering moving objects in a given scene can provide vital information about the trajectory patterns and outliers. The trajectory of an object may contain extended data at each position where the object was detected such as size, colour, *etc*.The focus of this work is to find an efficient trajectory clustering solution given the most fundamental trajectory data, namely position and time. The main challenge of clustering trajectory data is to handle the length of a single trajectory. The length of a trajectory can be extremely long in some cases. Hence it may cause problems to keep trajectories in main memory or it may be very inefficient to process them. Preprocessing trajectories and simplifying them will help minimize the effects of such issues. We will use some algorithms taken from literature in conjunction with some of our own algorithms in order to cluster trajectories in an efficient manner. In an attempt to accomplish this, we have designed a representation of a trajectory. Furthermore, we have designed and implemented algorithms to simplify and evaluate distances between these trajectories. Moreover, we proved that our distance function obeys triangulation properties which is beneficial for clustering algorithms. Our distance function is a variation of the Fréchet [11] distance proposed in 1906 by Maurice René Fréchet. Additionally, we will illustrate how our work can be integrated with an incremental clustering algorithm to cluster trajectories.

Acknowledgements

I would like to express my deepest appreciation to my supervisor Dr. Robert Laganière for supporting me and making it possible to complete this report. Words cannot express how grateful I am to him. It would be truly impossible to complete this work without his guidance and assistance. I would also like to thank the members of the committee for serving as my committee and taking the time to read my report and take part in my defence. I would also like to thank Dr. Prosenjit Bose and Dr. Pascal Blais for their support throughout my undergraduate and graduate studies.

A special thanks to my mother and my sister for supporting me throughout my academic career. I would also like to thank my father for being a role model in my life. It is hard not to have him around us anymore but I have to say that his influence in my life is beyond limits.

Last but not least, I would like to thank all my friends and classmates for encouraging me and supporting me all along the way. This would be impossible without any of you.

Contents

List of Symbols **xii**

1 Introduction **1**

1.1 Motivation 1
1.2 Problem Description 2
1.3 System Requirements 4
1.4 Clustering System Architecture 4
1.5 Contributions 6
1.6 Thesis Organization 7

2 Background **8**

2.1 Similarity Measures of Polygonal Curves 8
2.1.1 Hausdorff Distance 9
2.1.2 Fréchet Distance 11
2.1.3 Coupling Distance 12
2.2 Polygonal Chain Simplification 14
2.2.1 Approximation Error Measures 15
2.2.2 Early Dynamic Programming Approach 17
2.2.3 Computational Geometric Problem Formulation 18
2.2.4 Polygonal Chain Approximation based on Fréchet Distance 19

3 Trajectories in surveillance systems **24**

3.1 Trajectory Definition 24
3.2 Trajectory speed measurement 25
3.3 Last spatio-temporal node assumption 27
3.4 Frame-based Time Units 28
3.5 Time-Independent Trajectory 29

3.6 Conclusion 29

4 Trajectory Distance Function 31

4.1 Requirements of a Good Distance Measure 31
4.2 Trajectory Distance Function Definition 33
4.3 Metric Distance Function 34
4.4 Discrete Distance Function Optimality 39
4.5 Distance Function Algorithm 43
4.6 Conclusion 50

5 Trajectory Simplification 51

5.1 Purpose 51
5.2 Remote Trajectory Simplification Method 53
5.3 Approach 54
5.4 Doubling Search Method 55
5.5 Direct Link Simplification Method 60
5.6 Enclosing Disk Simplification Method 63
5.6.1 Smallest Enclosing Disk Problem 64
5.6.2 Randomized Solution 68
5.6.3 Enclosing Disk Simplification 72
5.7 Conclusion 75

6 Clustering 78

6.1 Introduction 78
6.2 Clustering Prerequisites 79
6.3 Hierarchical Agglomerative Clustering 80
6.4 BIRCH Clustering 82
6.5 BUBBLE Clustering for Metric Objects 86
6.6 FastMap Algorithm 92
6.7 BUBBLE-FM Clustering 94

7 Results 98

7.1 Dataset 98
7.2 Linear vs Quadratic distance function algorithms 99
7.3 Simplification Experiments 101
7.3.1 Running-Time Performance 101

7.3.2 Error Distribution 102
7.3.3 Size-Ratio Distribution 103
7.3.4 Conclusion 104
7.4 Clustering Experiments 106
7.5 Conclusion 107

8 Conclusion 111
8.1 Summary 111
8.2 Future Work 112
8.3 Conclusion 112

List of Tables

2.1 Coupling distance and discrete Fréchet distance relationship. 22

7.1 Dataset node count frequency 99
7.2 Scaled coordinates 99
7.3 Distance function running time. 101
7.4 Simplification method running time. 101
7.5 Total Size-Ratio Comparison 104
7.6 Dataset node range count 108
7.7 Clustering Running Time 108

List of Figures

1.1 Clustering methodology 3
1.2 Tracker Overview 5
1.3 Clustering system layers 6

2.1 Hausdorff Weakness as a Similarity Measure for Polygonal Chains 10
2.2 Fréchet distance 11
2.3 Coupling Distance 14
2.4 General Approximation Error 15
2.5 Undefined Projection 15
2.6 Additive Error 16
2.7 Parallel-strip (Infinite Beams) 16
2.8 Tolerance zone distance measure 17
2.9 Acyclic weight graph for min-# problem 18
2.10 Polygonal Chain Approximation 19
2.11 m-walk 20
2.12 paired-walk 20
2.13 Approximation closed under Fréchet distance. 22

3.1 Trajectory 25
3.2 Similar Trajectories 28

4.1 Parallel points moving in the same direction at the same speed 40
4.2 Parallel points moving in opposite directions 41
4.3 Parallel points moving in the same direction at different speeds 42
4.4 Points moving on non-parallel lines 43

5.1 Direct Link Simplification Example 55
5.2 Partitioned trajectory. 62

5.3 Uniqueness of Smallest Enclosing Disk 64
5.4 Enclosing Disk Simplification Example 74

7.1 Raw trajectories in a snapshot. 100
7.2 Disk simplified trajectories snapshot. 102
7.3 Direct-Link simplified trajectories snapshot. 103
7.4 Size-Ratio Frequency Comparison 105
7.5 Trajectories after enclosing disk simplification in a snapshot. 106
7.6 Raw trajectories in a snapshot. 107
7.7 Sample Clusters 109
7.8 Sample Clusters 110

List of Symbols

$P = (p_1, p_2, ..., p_n)$	Polygonal Chain	8
$S = (M, d)$	Metric Space	9
δ_H	Hausdorff Distance	9
δ_F	Fréchet Distance	12
L	Coupling	13
δ_{dF}	Coupling Distance	13
$\{\mathcal{P}\}_{i=1..m}$	m-walk	19
$d_F^W(P, Q)$	Cost of Paired Walk	21
$d_F(P, Q)$	Discrete Fréchet Distance	21
$c = (p, t)$	Spatiotemporal Node	24
$T = (c_1, c_2, ..., c_n)$	Trajectory	25
$\alpha_T(t)$	Trajectory Time Function	34
$d_{PQ}(t)$	Time-Distance Function	34
$\delta_T(P, Q)$	Trajectory Distance Function	35
$\delta_{dT}(P, Q)$	Discrete Trajectory Distance Function	39
$f_R(L, k)$	Abstract Rule Function	56
$f_{dl}(T, i)$	Direct Link Rule Function	61
$f_{disk}(T, i)$	Enclosing Disk Rule Function	75
$d_{SL}(G, H)$	Single Linkage Measurement	81
$d_{CL}(G, H)$	Complete Linkage Measurement	81
$d_{GA}(G, H)$	Group Average Measurement	81
$\vec{X}$	Centroid of a Set of Vectors	83
R	Radius of a Set of Vectors (BIRCH)	83
D	Diameter of a set of vectors	83
$D0$	Centroid Euclidean Distance (BIRCH)	83
$D1$	Centroid Manhatan Distance (BIRCH)	83
$D2$	Average Inter-Cluster Distance (BIRCH)	84

$D3$	Average Intra-Cluster Distance (BIRCH) 84
$D4$	Variance Increase Distance (BIRCH) 84
$RowSum(o)$	RowSum of a Set of Metric Objects 88
$\hat{O}$	Clusteroid 88
R	Radius of Set a Metric Objects(BUBBLE) 89
$D0$	Clusteroid Distance(BUBBLE) 89
$D2$	Average Inter-Cluster Distance (BUBBLE) 89

Chapter 1

Introduction

This thesis presents a technique to simplify, compare and cluster trajectories in an R^d plane. There are two main factors that must be taken into account when comparing trajectories. The first factor is the path of the object and the second factor is the velocity at which an object moves through space. Clustering is performed after preprocessing the trajectories using a simplification algorithm introduced here. The simplification speeds up the computation of the distance between trajectories which leads to a faster clustering algorithm. The approach presented here can can be applied in a real-time scenario.

1.1 Motivation

The emergence of fast multi-core processors in the recent decade has made it possible for a numerous computer vision algorithms to evolve that once were not practical. Additionally, the increased availability of video technology has motivated researchers on the problem of video object tracking. Surveillance systems are used in a wide range of environments such has shopping malls, parking lots, industrial environments, military establishments. Tracking an object in space provides a tremendous amount of data with respect to the observed scene. Data such as the speed at which objects move and the path of the objects may further be used. Learning about objects movement patterns can be very useful for a lot of applications such as traffic light synchronization. Surveillance cameras may also benefit from information about these patterns. In the case of surveillance systems, one may use object's movement patterns to distinguish between the typical paths of people or cars versus a unique path of an object. Such information may easily be extracted if the objects movements are clustered in groups.

In particular, taking advantage of such a system in order to synchronize traffic lights may replace older technologies with a much cheaper solution. Traffic lights are used in almost all countries across the world and cities spend lots of money to maintain them. However the maintenance cost of traffic lights is not the only expense for the city. In order to synchronize the lights depending on the traffic, pavement sensors are installed to collect data with respect to traffic conditions. These sensors are called ”inductive-loop traffic detectors” [19]. This data is further analyzed to synchronize traffic lights. The fact is that such systems are not only expensive to purchase and install but also they need to be maintained. In particular the cost can be exponential in the case of a need to replace a sensor due to its failure.

On the other hand, the cost of surveillance cameras have dropped over the last decade with the rise of new technologies. It is a cheaper solution to replace pavement sensors with cameras if the traffic conditions can be monitored using a simple camera. In order to monitor the traffic, the system must have the ability to track the movement of cars. Clustering objects movements can provide information about the frequency of objects moving from one direction to another. Such information can further be analyzed to interpret the traffic condition and synchronize traffic lights.

This clustering system can be beneficial when used in other applications as well. It may be a hard task for a human to observe a crowded scene in order to detect suspicious activities. Clustering movements of people can make it easier to look into unique movement patterns. The system can then notify the user about a unique movement in the crowd. Once such information is provided to the user, it becomes an easier task to identify suspicious activities.

In fact, clustering movements of objects can be applied to a whole array of problems in general. We have covered two particular examples so far. However, depending on how the data is collected, the applications may change. For example, biologists who study the migration patterns of birds and other animals may collect trajectory information by attaching positioning devices to the animals. Once the data is collected, clustering animals movements may provide them with a good visualization of animals’ migration patterns.

1.2 Problem Description

The problem is the following. We want to design a system that takes a set of 3D trajectories of objects as input and cluster them such that similar trajectories end up in the same group together.

In general, we do not wish to make any assumptions as to how the input to our system was collected. However, in order to perform our experiments, we had to target a specific source of data. We had developed and implemented a surveillance tracking algorithm in the past at the

VIVA Lab which we used to collect our data. Our tracking algorithm was based on a model-free algorithm [26] with some of our own enhancement to it. The main modification to the algorithm was the incorporation of a Kalman filter [17] as a predictor of object's position.

In order to ensure that the system can be applied in a real-world environment, we would like the clustering algorithm to satisfy two main properties; (1) The clusters in the system must incrementally evolve. (2) The system must be smart to make use of memory and processor resources appropriately.

The former property is there to ensure that the system can be applied in a real-time scenario. For instance, take the traffic light synchronization problem for example. In such a scenario, the system must be adaptive to changes in the traffic conditions in real-time to synchronize traffic lights appropriately. The latter property is there to ensure that the system can independently run for a long time. The latter property is crucial because a single trajectory may need a lot of memory resources. Keeping all trajectories in main memory will cause the system to run out of memory. It will also cause the system to slow down as more trajectories are inserted into each cluster.

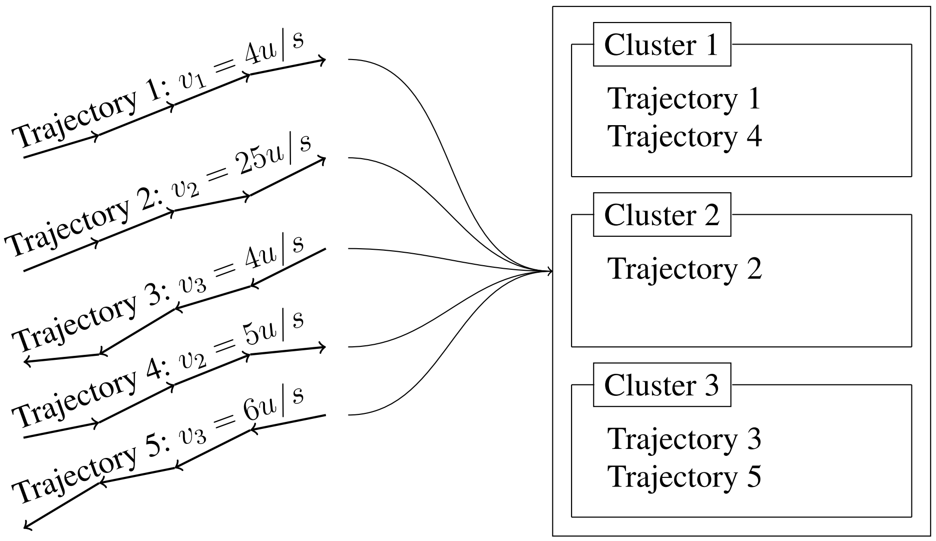
Trajectory 1: $v_1 = 4u|s$
Trajectory 2: $v_2 = 25u|s$
Trajectory 3: $v_3 = 4u|s$
Trajectory 4: $v_2 = 5u|s$
Trajectory 5: $v_3 = 6u|s$
Cluster 1
Trajectory 1
Trajectory 4
Cluster 2
Trajectory 2
Cluster 3
Trajectory 3
Trajectory 5

Figure 1.1: Clustering methodology. In this figure, v_i is the velocity of i^{th} trajectory in units/sec.

Figure 1.1 illustrates the main idea behind the target clustering system. We have five trajectories added incrementally into the clustering system. Given this input, three clusters evolved through the clustering system. Imagine the bottom left corner of all of the 5 trajectories be relatively around the same area. Notice trajectory 2 did not end up in cluster 1. This is due to the variation of the velocity. Trajectories 1 and 4 move at almost the same speed *i.e.*, 4 units per second and 5 units per second respectively. Conversely, trajectory 2 moves at speed of 25 units per second which is much higher than those of trajectories 1 and 4. Therefore, even though trajectory 2 takes a similar path to 1 and 4, they will be grouped in different clusters. Similarly, trajectories 3 and 5 take a very similar path and they move almost at the same speed therefore they both end up in the same cluster. Figure 1.1 is only used to illustrate a general idea behind the clustering system. However, it may be misleading because it may suggest that we are making the assumption that objects move at a constant speed. We will see that objects can slow down or speed up and therefore these variations will be taken into account.

1.3 System Requirements

The aim of this work is to create a system to identify trajectory's of moving objects and assign them to proper clusters in real time. Identification of moving objects in a crawdad scene can be a very expensive operation by itself. The identified trajectory's must be stored in a storage device or main memory in some type of a data structure. We need to define minimum system requirements that could operate such a system in real time. The real minimum requirements depends on the defined parameters.

This clustering system was implemented and operated on a laptop with the following specifications:

- AMD Quad-Core Processor A8 with Turbo CORE Technology up to 3.1 GHz
- 8GB DDR3 Memory
- 500GB Hard Drive

On top of these specifications, MySQL v5.0 was used to store some of the identified trajectory's in the storage device. With these speficiations, this clustering algorithm operates in real-time.

1.4 Clustering System Architecture

In order to start our research, we needed a good dataset containing a set of trajectories. We used a tracking algorithm to capture trajectories from a fixed-position camera. Each trajectory represents an object moving in the scene. Each trajectory is an ordered sequence of nodes. Each node of the trajectory in the original dataset represents the position of the object at the time of capture. Additionally, the i^{th}node of the trajectory contains extra information about the object: (1) The location of the object denoted by $p_i \in \mathrm{R}^2$ (2) The time t_i when the object was captured at that position (3) A 16 dimensional vector $\vec{C} \in \mathrm{N}^{16}$ representing the colour histogram of the silhouette of the object. (4) The centroid of the silhouette of the object denoted by $c \in \mathrm{R}^2$.

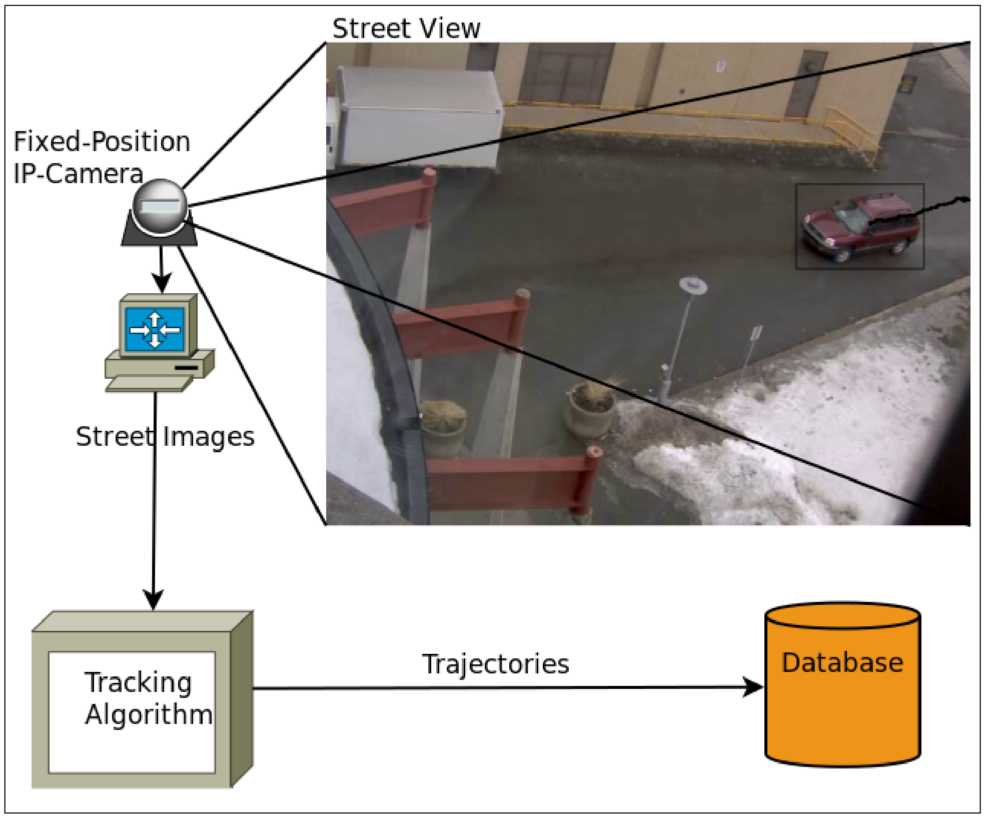

Figure 1.2: Tracking system overview

Even though lots of information were associated with each trajectory, we only use the spatio-temporal data of each node for clustering. Our fixed-position camera along with the tracker algorithm gathered trajectory data for nearly a month. The data was added to a database while being gathered from the system. Figure 1.2 illustrates how images were captured and added to the database.

This off-line architecture was deployed in order to ensure the possibility of testing our algorithms repeatedly over the same dataset. Once enough data is gathered, it is fed into our clustering system. The clustering system consists of two main layers: (1) Simplification layer. (2) Clustering layer. Figure 1.3 illustrates how these layers interact with each other.

The simplification layer reads trajectories from the data as input in an incremental fashion.

Each trajectory is simplified to reduce the number of nodes of the trajectory. The simplified trajectory is then sent into the clustering layer. The clustering layer uses our distance function δ_T and a clustering algorithm BUBBLE [12] to incrementally evolve clusters. Throughout this process, clusters can be observed by the user through a different thread. This architecture can

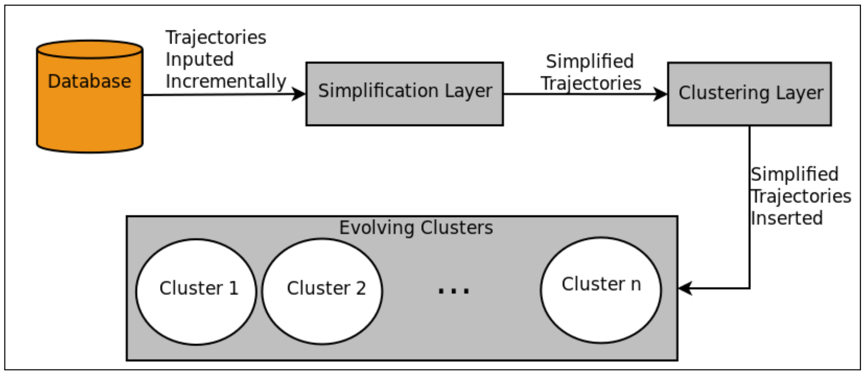

Figure 1.3: Layers of the clustering system

be modified to a real time scenario where the trajectories are directly inputted from the tracking system. This is due to the fact that the average number of clusters in a frame of time generated by the tracking algorithm is too small to disrupt the clustering system. This can be achieved by a replacing the database by a trajectory queue in our architecture.

1.5 Contributions

The contributions of this system include a clustering algorithm to group similar trajectory's moving objects into groups. These clusters can be used to identify suspicious activities in a surveillance system. Such a system can be used to mark a cluster as suspicious. This can help label prospective trajectory's as suspicious if they are placed in a suspicious cluster. Furthermore, the system can alert users if a trajectory starts a new cluster. This means that the system was unable to find a similar cluster for such a trajectory. The user can then review the trajectory and mark it as suspicious or normal.

The main contributions of this thesis is the introduction of a trajectory dissimilarity measure which we define as a distance function. This distance function is a variation of Fréchet distance that also accounts for time and velocity. We prove that δ_T obeys the triangulation property. Hence, it is viable to use it in conjunction with any metric-based clustering algorithm. Additionally, we also introduce an algorithm to simplify input trajectories. This simplification is performed with respect to a user-defined threshold ϵ. Given a trajectory A, our simplification algorithm outputs a new trajectory A' such that $\delta_T(A, A') \leq \epsilon$. This algorithm is used as a preprocessor to improve the processing time of the clustering algorithm. We show how our work can be integrated with a variant of an incremental clustering algorithm BUBBLE-FM [12] to cluster trajectories in real time. Additionally, the application of Fréchet distance to cluster trajectories from surveillance data is also a contribution.

1.6 Thesis Organization

Chapter 1 introduces the problem and provides an overall picture of the system we are trying to design.

Chapter 2 reviews the past research relative to this thesis. It covers different techniques for comparing trajectories and simplifying them. It also introduces the Fréchet distance measurement which is a solid foundation of this thesis.

Chapter 3 covers the fundamental mathematical definitions that are used throughout this thesis. It formally defines trajectories. It also covers all the fundamental assumptions made in this thesis regarding trajectories.

Chapter 4 covers the mathematical definition of our distance function. This distance function is our main tool to compare trajectories against each other. It also covers some proofs that are required by our clustering algorithm. Chapter 4 also covers a very efficient algorithm that is used to compute our distance functions.

Chapter 5 covers the simplification approach used in this thesis. It also covers why simplifying trajectories is a good measure to take before clustering process starts. Chapter 5 also covers the effect of simplification on the quality of the clusters. It also introduces our simplification algorithms in details.

Chapter 6 covers the clustering algorithms used in this thesis. It covers some different clustering algorithms and discusses why we prefer some particular approaches over others.

Chapter 7 covers the experiments we run to cluster trajectories.

Chapter 8 concludes this thesis.

Chapter 2

Background

Trajectories are a unique type of polygonal chains. A polygonal chain is a connected series of line segments.

Definition 2.1 (Polygonal Chain). A *polygonal chain P is a curve specified by a sequence of points $(p_1, p_2, ..., p_n)$ called vertices such that for some $1 \leq i < n$, vertices p_i and p_{i+1} are connected by an edge.*

Our formal trajectory definition is given in Chapter 3. Informally, a trajectory is the same as a polygonal chain with one more piece of information. Each trajectory vertex is associated with a time stamp. Since polygonal chains are very similar to trajectories, they seem to be a good starting point. Recall Chapter 1 introduced trajectory simplification as the first layer of the trajectory clustering system. In this chapter we present a review of related work in polygonal chains and techniques used to compare them. We will also review polygonal chain simplification.

2.1 Similarity Measures of Polygonal Curves

Polygonal chains are well-studied in computational geometry. This is due to their natural properties and they show up in various problems in CG such as the classical shortest path problem. They can be exploited to describe a whole range of data such as the path of a packet taken in a network model or the direction a car needs to take in a GPS device. Traditionally, polygonal chains have been a subject of study for mathematicians. A typical approach to clustering problems in computer science is to have a mathematical representation of the objects of interest. Furthermore, one needs to define a distance function between such objects. This distance function is referred as the similarity measure between a pair of objects. Maurice René Fréchet introduced a very famous distance function between polygonal chain in 1906 [11]. In fact this approach has been used extensively in computer science research. The Hausdorff distance [14] introduced in 1914 by Hausdorff is a similarity measure between finite sets in the same metric space. The Hausdorff distance has also been widely used as a similarity measure to compare polygonal chains. The Discrete Fréchet distance [8] introduced by Eiter and Mannila in 1994 is a variation of the Fréchet distance. The Discrete Fréchet distance is easier and faster to compute than the original Fréchet distance. This section reviews these similarity measures.

2.1.1 Hausdorff Distance

The Hausdorff distance [14] is a similarity measure of two subsets of a metric space. The Hausdorff distance may be defined informally as follows. Two subsets of a metric space are close in the Hausdorff distance if every point of either subset is close to a point of the other subset. In order to formally define Hausdorff distance, it is worth defining metric spaces.

Definition 2.2 (Metric Space Ordered Pair). A *metric space S is an ordered pair (M, d) where M is a set and d is a distance function on elements of M:*
$d : M \times M \to \mathrm{R}$
such that for any $x, y, z \in M$, the following properties are satisfied:
(1) $d(x, y) \geq 0$
(2) $d(x, y) = 0 \leftrightarrow (x = y)$
(3) $d(x, y) = d(y, x)$
(4) $d(x, z) \leq d(x, y) + d(y, z)$

One of the most intuitive metric spaces is the pair of points in R^3 and the Euclidean distance function between points in R^3. We may formally define the Hausdorff distance as follows:

Definition 2.3 (Hausdorff Distance). L*et X and Y be two non-empty subsets in a metric space $S = (M, d)$. The Hausdorff Distance $\delta_H(X, Y)$ is defined as:*

$$\delta_H(X, Y) = \max\left\{\sup_{x \in X} \inf_{y \in Y} d(x, y), \sup_{y \in Y} \inf_{x \in X} d(x, y)\right\}$$

Intuitively, the Hausdorff distance can be calculated as follows. Consider a metric space $S = (M, d)$ and two subsets of M: X and Y. We take each element of X and compute its distance with all the elements in Y. We keep track of the minimum distance of each element in X to all elements in Y. We let a be the maximum of the tracked distances. Moreover, the same processing is done again but this time we switch the two subsets X and Y. This time, we let b

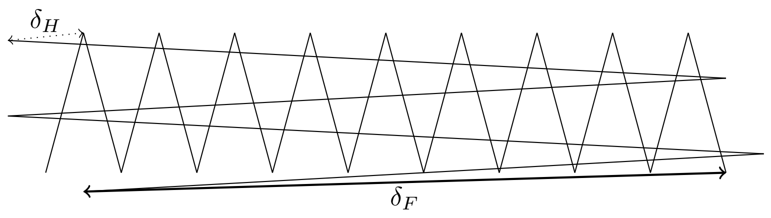

Figure 2.1: An example where Hausdorff distance does not properly capture path differences between polygonal chains. In this figure, the dotted line illustrates the Hausdorff distance and the thick line illustrates the Fréchet distance.

be the maximum of the tracked distances. The Hausdorff distance is equal to the maximum of a and b.

The Hausdorff distance is used in computer vision to find a given template in an arbitrary image. The template image and the target image will be preprocessed using an edge detector. Edge detectors such as Canny edge detector [3] result in a binary image where activated pixels represent edges. Furthermore, each activated point in the binary image of the template is treated as one subset. Similarly, activated points in a region of the binary target image, known as region of interest, is treated as the second subset. The algorithm scans the image for such regions of interest and for each region, it calculated the Hausdorff distance against the template binary subset. The region with the minimal distance will be the best candidate for locating the template object.

The question is whether Hausdorff Distance can be used as a similarity measure between polygonal chains. The answer is affirmative. Let A and B be two subsets of R^2 such that A contains all the points on one polygonal chain and B corresponds to the points of the other chain. One can simply compute the similarity measure of the two curves by computing $\delta_H(A, B)$. The Hausdorff distance mostly captures shape similarities. The second question is whether Hausdorff distance is an appropriate similarity measure to capture differences in paths taken by two polygonal chains. The answer to the second question is negative and we can show this by a simple example. Consider the two curves in Figure 2.1. Clearly the Hausdorff distance between the two curves is very small. However, the paths taken by the two curves are very different. We look further to find a better similarity measure that could capture path differences between polygonal chains.

2.1.2 Fréchet Distance

The Fréchet Distance [11] can be intuitively defined as follows. Imagine a man and a dog on a leash: The man moves on one curve and the dog moves on the other curve. Both the man and the dog are allowed to move on their corresponding curves at varying speeds but backtracking is not allowed. The length of the shortest leash needed for traversing both curve is the Fréchet distance between the two curves.

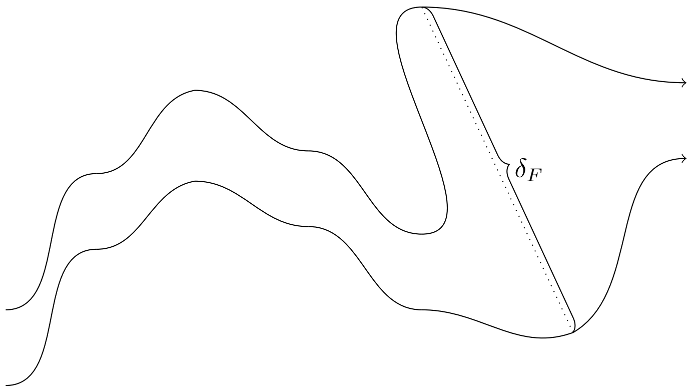

Figure 2.2: Illustrates the Fréchet distance between two curves.

The formal definition of the Fréchet Distance depends on two other definitions: (1) definition of a curve in a metric space; (2) reparameterization mapping. A curve in metric space is as follows:

Definition 2.4 (Metric Space Curve). L*et S be a metric space. Curve A is defined as a continuous map from unit interval into S i.e., $A : [0, 1] \to S$.*

Definition 2.4 may be non-intuitive. We give an example in R^2. Imagine a curve in R^2 coordinate system. The length of the curve could be anything. The curve can be defined as a mapping from a variable a to a point $p \in \mathrm{R}^2$. Variable a takes a value from 0 to 1 and it represent the length of the curve traversed. Point p is a point on the curve. For example, when $a = 0.5$, p is the point positioned exactly at the middle of the curve and when $a = 0$, p is the head of the curve and when $a = 1$, p is at the end of the curve.

Reparameterization mapping is defined as follows:

Definition 2.5 (Reparameterization Mapping). R*eparameterization α of $[0, 1]$ is a continuous, non-decreasing, surjective function $\alpha : [0, 1] \to [0, 1]$.*

Finally we may formally define the Fréchet distance:

Definition 2.6 (Fréchet Distance). L*et P and Q be two metric curves in the metric space S. Let α and β be two reparameterization mappings. Furthermore, let d be a distance function in the metric space S. The Fréchet distance δ_F is defined as:*

$$\delta_F = \inf_{\alpha,\beta} \max_{t\in[0,1]} \{d(P(\alpha(t)), Q(\beta(t)))\}$$

In Definition 2.6, t is the measure of time. The two reparameterization mappings α and β control the speed of P and Q at time t. Let P and Q correspond to the path to which the person and the dog walk on respectively. It is worth noting that the Fréchet distance is used in several problems such as morphing [7], handwriting recognition [29], protein structure alignment [16] and much more. Additionally, from the informal definition of the Fréchet distance, it is clear it can be used a similarity measure between polygonal chains. If the length of the shortest leash needed for the person and the dog is very large the paths taken by the dog must be very different than the path taken by the person. The Fréchet distance considers all possible speeds at which both objects move through space but trajectories have a unique speed at each point on their path. We conclude that the Fréchet distance can be used as a base in search of a proper similarity measure between trajectories but some variations of it is required to capture speed differences.

2.1.3 Coupling Distance

The Fréchet Distance is a powerful mathematical similarity measure. The first fundamental study on computation of Fréchet distance was done by Alt and Godau [1] in 1992. They introduced the decision problem of Fréchet distance: Let P and Q be two polygonal curves and $\epsilon \in \mathrm{R}$. The problem is to decide $\delta_F(P, Q) \leq \epsilon$. They showed how to solve the decision problem. Furthermore, they showed how to use the solution to the decision problem to compute the exact Fréchet distance. Alt and Godau [1] designed an algorithm that computes the Fréchet distance in $O(pq \log^2 pq)$ where p and q are the number of line segments on the polygonal curves respectively. Their algorithm is fairly complex due to the usage of a parametric search technique. This raises two issues for a trajectory clustering algorithm: (1) Computing the Fréchet distance is computationally expensive for a clustering algorithm dealing with a large dataset. (2) It is quite complex to modify the algorithm after introducing a variation of the Fréchet distance that accounts for speed of the objects. Hence, we resume our search for a simpler and faster method. Eiter and Mannila [8] introduced a discrete variant of the Fréchet distance in 1994. They called this variation the coupling distance denoted by δ_{dF}. The algorithm is based on all possible couplings between the end points of the line segments of the polygonal curves. The coupling distance is an approximation of the Fréchet distance. In fact the coupling distance is an upper bound for the Fréchet distance. The difference between these two similarity measures is bounded by the length of the longest edge of the two polygonal chains. The coupling distance can be computed in $O(pq)$ time. Additionally, the algorithm introduced by Eiter and Mannila to compute the coupling distance is very simple. This section gives an introduction to δ_{dF} and the algorithm proposed by Eiter and Mannila for computing the coupling distance.

We first define a couple L between two polygonal curves as follows:

Definition 2.7 (Coupling). L*et* $P = (u_1, u_2, ..., u_p)$ *and* $Q = (v_1, v_2, ..., v_q)$ *be two polygonal curves. A coupling L between P and Q is defined as a sequence:*

$$(u_{a_1}, v_{b_1}), (u_{a_2}, v_{b_2}), ..., (u_{a_m}, v_{b_m})$$

Such that:
(1) $a_1 = 1, b_1 = 1, a_m = p, b_m = q$.
(2) For all $i = 1, ..., m$*, we have* $(a_{i+1} = a_i \oplus a_{i+1} = a_i + 1) \wedge (b_{i+1} = b_i \oplus b_{i+1} = b_i + 1)$.

The two conditions defined in Definition 2.7 ensure that all the vertices of the two curves are covered in an increasing order. The first condition makes the case such that the index range of both polygonal chains are from the first index to the very last index. The second condition ensures that no index is skipped and the sequence is in an increasing order.

Furthermore, the length of a coupling denoted by $\|L\|$ is defined as follows:

Definition 2.8 (Length of coupling L). T*he length of a coupling L is the length of the longest link in L:*

$$\|L\| = \max_{i=1,...,m} d(u_{a_i}, v_{b_i})$$

Finally, we may introduce the coupling distance between P and Q:

Definition 2.9 (Coupling Distance). G*iven two polygonal curves P and Q, their coupling distance is defined as the minimum coupling distance of all possible couplings of P and Q:*

$$\delta_{dF}(P, Q) = \min \{\|L\| \mid L \textit{ is a coupling of } P \textit{ and } Q\}$$

Clustering algorithms such as BUBBLE [12] and DBSCAN [9] depend on a metric on the set of objects to be clustered. An important advantage of using the Coupling distance is that it defines a metric on the set of polygonal curves. Eiter and Mannila show this in [8]. The other reason to consider the Coupling distance is the simplicity and running time of the algorithm

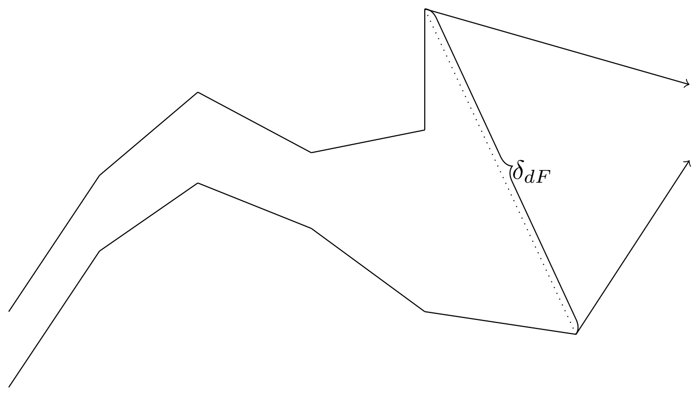

Figure 2.3: The image above illustrates the coupling distance between two discrete curves.

to compute the distance. Additionally, if the polygonal chains do not have very long edges then the Coupling distance is very close to the Fréchet distance which can fully capture the differences between the paths taken by the polygonal curves.

2.2 Polygonal Chain Simplification

Polygonal chains can contain a large number of points. This can make distance computation of polygonal chains a time consuming process. Clustering algorithms make numerous calls to these distance functions and this quickly becomes problematic even for small datasets. Similar problems arise in many applications including geographic information systems (GIS), computer graphics and data compression. Therefore, a lot of attention has been given to such problems among researchers. In order to formulate the problem, we first need to give a definition to the polygonal chain approximation:

Definition 2.10 (Polygonal Chain Approximation). L*et* $P = (p_1, p_2, ..., p_n)$ *be a polygonal chain and an error bound* ϵ *then a subchain* $P' = (p_{i_1}, p_{i_2}, ..., p_{i_m})$ *is an* ϵ*-approximation of* P *with* $1 = i_1 < i_2 < ... < i_m = n$ *such that the error of each segment* $p_{i_l}p_{i_{l+1}}(l = 1, ..., m-1)$ *is at most* ϵ*.*

The approximation error is defined appropriately according to the respective application in Definition 2.10. The min-# problem is defined as: Given a polygonal chain P and an error bound ϵ, compute an ϵ-approximation of P with the minimum number of vertices. Subsection 2.2.1 introduces a few common approximation error measures used in literature.

2.2.1 Approximation Error Measures

The error criterion is purposely left undefined in Definition 2.10. That is because the error criterion must be defined appropriately for a target application.

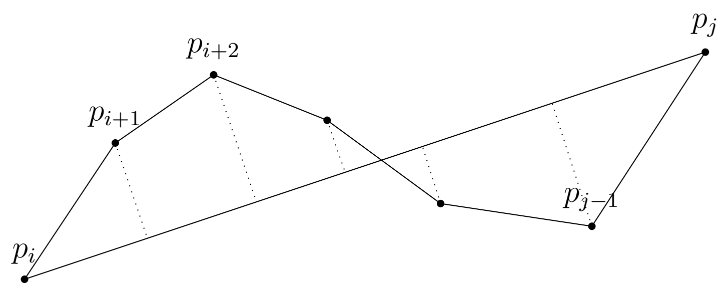

Figure 2.4: The error of the line segment P_iP_j is based on distances between the vertices $P_i, P_{i+1}, ..., P_j$ and line segment P_iP_j

In practice, error measures are normally based on distance between vertices of an input curve and the approximation linear segments. In Figure 2.4, the projections of the points $P_i, ..., P_j$ onto the line segment P_iP_j are all defined. However, these projections can be undefined (Figure 2.5) and different error measures define different rules to deal with such points.

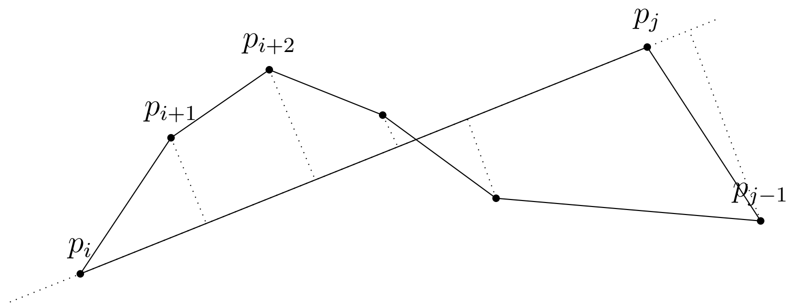

Figure 2.5: In the example above, no projection of p_{j-1} onto the line segment p_ip_j exists. In such cases different alternatives are taken by various error measures. One solution is to consider the projection of the point onto the line that passes through the points p_i and p_j. Another alternative is to compute the shortest distance between the point to the line segment. In the example above, it would be the distance between p_{j-1} to p_j.

One of the error measures considered in literature is the additive error measure. That is the sum of distances between the points $p_i, ..., p_j$ to the line passing through p_i and p_j(Figure 2.6). Let $P_{ij} = (p_i, p_{i+1}, ..., p_j)$ be a sub-curve of P. Furthermore, for all $i \leq l \leq j$, let p_l' be the projection of p_l onto the line passing through p_i and p_j. The additive error measure of the line segment p_ip_j is defined as:

$$e_{additive}(i,j) = \sum_{l=i}^{j} \|p_lp_l'\| \tag{2.1}$$

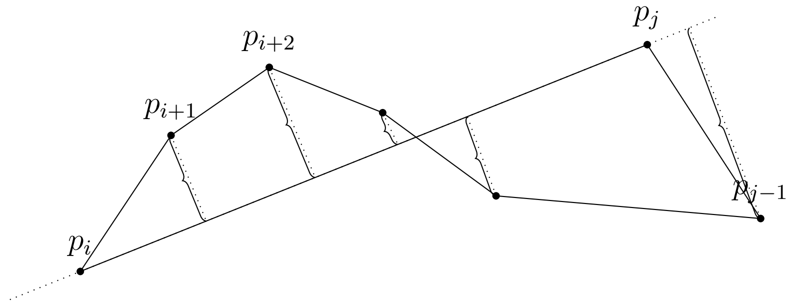

Figure 2.6: In the example above, the additive error measure is the sum of the distances shown with the curly brackets.

Perhaps one of the most common error measures used among researchers is the parallel-strip criterion. The parallel-strip is the maximum distance between the points $p_i, ..., p_j$ to the line passing through p_i and p_j(Figure 2.7). Let $P_{ij} = (p_i, p_{i+1}, ..., p_j)$ be a sub-curve of P. Furthermore, for all $i \leq l \leq j$, let p_l' be the projection of p_l onto the line passing through p_i and p_j. The parallel-strip error measure of the line segment p_ip_j is defined as:

$$e_{parallel}(i,j) = \max_{l=i,...,j} \{\|p_lp_l'\|\} \tag{2.2}$$

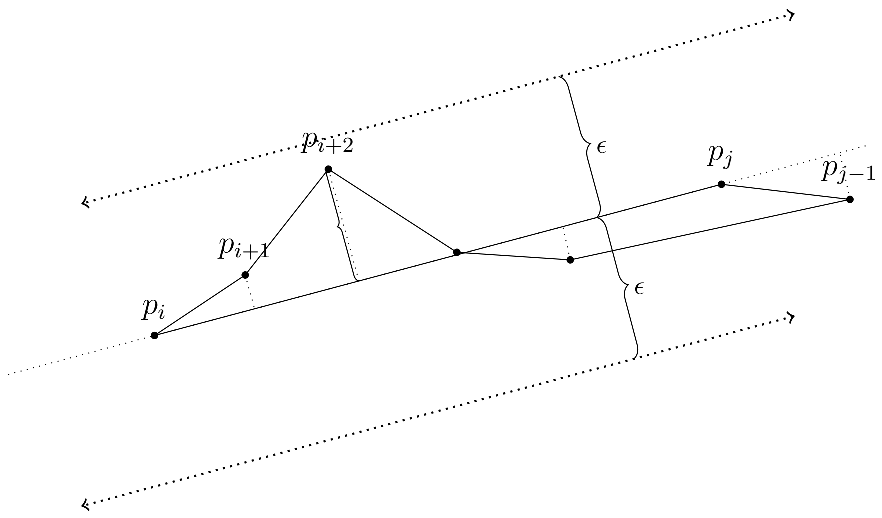

Figure 2.7: In the example above, the parallel-strip distance of the line segment p_ip_j is the distance between p_{i+2} and its projection on the line segment.

Parallel-strip error is also known and the infinite beam criterion. Consider two infinite beams parallel to the line segment P_iP_j exactly ϵ distance away from both directions of the line segment. As long as all the points $P_i, ..., P_j$ stay between these two parallel beams, the approximation of the line segment P_iP_j is valid.

Another common error criterion used in the literature is the tolerance zone error measure. This error measure is very similar to the infinite beam but it considers the shortest distance between points and the line segment p_ip_j. Let $P_{ij} = (p_i, p_{i+1}, ..., p_j)$ be a sub-curve of P. Furthermore, for a point $q \in \mathrm{R}^d$, let $d_{ij}(q)$ be the minimum distance between point q to the line segment p_ip_j. The tolerance zone error measure of the line segment p_ip_j is defined as:

$$e_{tolerance}(i,j) = \max_{l=i,...,j} \{d_{ij}(p_l)\} \tag{2.3}$$

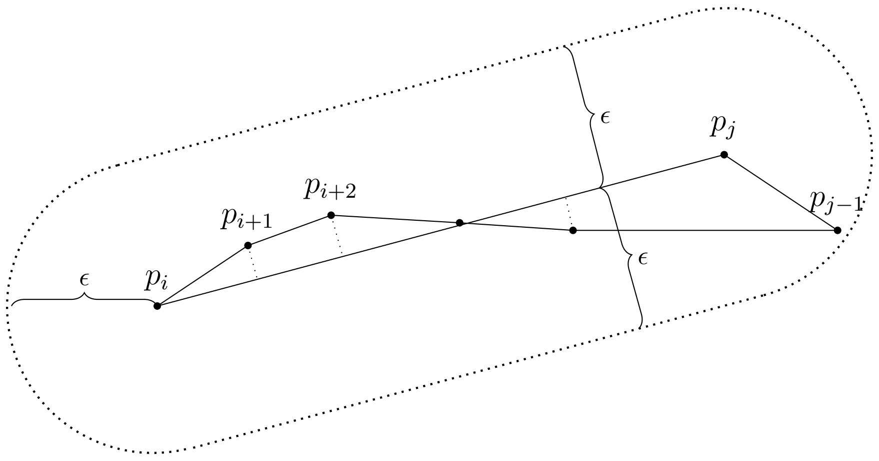

Figure 2.8: In the example above, the tolerance zone error of the line segment p_ip_j is the distance between p_j and p_{j-1}

Consider an ϵ-approximation of the line segment p_ip_j. The approximation is only valid if all the points $p_i, ..., p_j$ exist in a region around the line segment such that the boundary of the region is ϵ distance away from the line segment. Figure 2.8 illustrates this boundary for a line segment p_ip_j.

2.2.2 Early Dynamic Programming Approach

One of the earliest solutions to the min-# problem was the dynamic programming approach introduced by Papakonstantinou [27] back in 1985. This algorithm would optimize the number of line segments allocated to different individual curves. The tolerance region was defined by the infinite beam criterion also know as Parallel-strip: the maximum distance between the line p_ip_j and the points of the curve segment $\{p_i, ..., p_j\}$.

The idea was to optimize the number of line segments iteratively and the attempt was to reduce the number of line segments of sub-curves at each iteration. The algorithm could iterate infinitely. Hence, the trade off between time and the optimality of the approximated curve would be controlled by the breadth of the search and the number of iterations applied.

2.2.3 Computational Geometric Problem Formulation

In 1986, Imai and Hiroshi [15] presented a unified approach to formulate the min-# problem in terms of graph theory. They considered a directed graph G of which each vertex v_i represents a point P_i from the input sequence. Furthermore, the graph contains edges (v_i, v_j) where $i < j$ if and only if the error of the segment is at most w'. The segment errors can be any of the error criterion. Graph G is also weighted and each edge (v_i, v_j) is assigned a weight of $j-i-1$ which is the number of segments that can be skipped if the points P_i and P_j are to be connected. The approximate curve with the minimum number of edges is the longest path from v_1 to v_n where the length is measured by the weights of the edges. Observe that the graph G has no cycles and therefore it is an acyclic weighted graph. Hence, the longest path from v_1 to v_n can be found in time proportional to the number of the edges in the graph G which is $O(n^2)$. The bottleneck of this algorithm is generating the graph G which takes $O(n^3)$ time if a naive approach is taken. However, O'Rourke and Melkman [25] showed how to construct the graph in $O(n^2 \log n)$ and therefore the problem can be solved in time $O(n^2 \log n)$. Figure 2.9 demonstrate an example of the graph G based on the input set of points P and the tolerance zone error criterion.

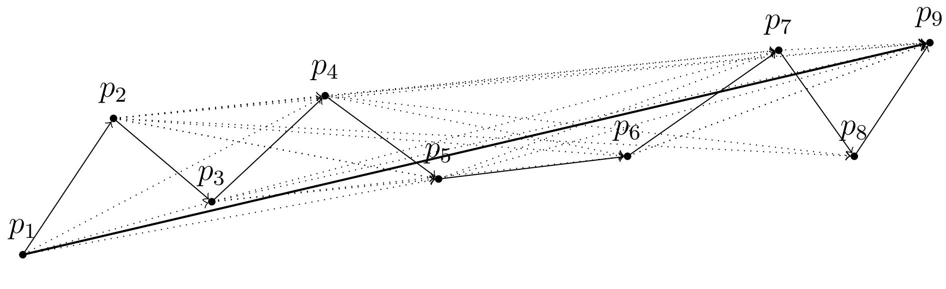

Figure 2.9: The input polygonal chain contains of 9 points $p_1, ..., p_9$. All the lines are edges of the graph G. The single thick line between p_1 to p_9 is the simplified polygonal chain.

Later in 1992, Chan and Chin [5] improved the algorithm so it takes $O(n^2)$ time to construct the graph using the parallel beam criterion. Theoretically, this algorithm would be the optimal solution for the graph-based formulation because the graph G could have $O(n^2)$ edges in the worst case and it is impossible to implement a faster algorithm.

2.2.4 Polygonal Chain Approximation based on Fréchet Distance

We have seen the min-# problem which is based on a given error criterion. The min-# problem seeks a solution where the approximation contains a subset of the points of the given input polygonal chain. However, this seems to be a very strong assumption. Another way to look at the approximation problem is to consider a solution where the points of the approximation of a chain are near the ones in the original input chain and not necessarily a subset of them. In Figure 2.10, the two polygonal chains P shown in black and Q shown in red do not share the same points but they take a very similar path. Moreover, Q can be thought of a simplified version of P. This gives rise to a new formulation of the approximation problem.

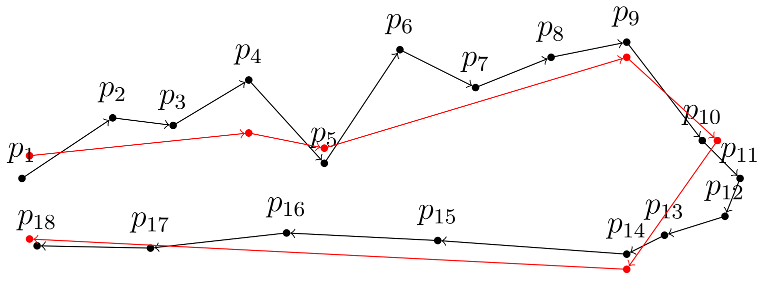

Figure 2.10: Polygonal chain P has 18 points and the chain Q shown in red has only 8 points. Notice Q and P do not share the same points.

We can informally introduce the problem: Given a chain A with n points in a d-dimensional space, compute a polygonal chain A' whose number of points are much smaller than of n and A and A' share a very similar path. We have already introduced the notion of discrete Fréchet distance. In order to give meaning to what describes a similar path, we may use discrete Fréchet distance. The problem can be formally define as follows: Given a polygonal chain A in a d-dimensional space with n vertices and an error bound ϵ, compute a polygonal chain A' with the minimum number of vertices such that $\delta_{dF}(A, A') \leq \epsilon$. Bereg and Jiang [2] gave a solution to this problem that runs in $O(n \log n)$ time. This section gives an overview of their method. In order to explain their methodology, one needs to think of the discrete Fréchet distance at a different angle. Let us first define the notion of an m-walk and a paired walk:

Definition 2.11 **(m-walk).** *Given a polygonal chain $P = (p_1, p_2, ..., p_k)$ of k vertices, an m-walk along P partitions the path into m disjoint, non-empty subchains $\{\mathcal{P}_i\}_{i=1..m}$ such that*

$\mathcal{P}_i = (p_{k_{i-1}+1}, ..., p_{k_i})$ *and* $1 = k_0 < k_1 < ... < k_m = k$.

Definition 2.12 (Paired walk). *Given two polygonal chains* $P = (p_1, p_2, ..., p_k)$ *and* $Q = (q_1, q_2, ..., q_k)$*, a paired walk along* P *and* Q *is an* m*-walk* $\{\mathcal{P}\}_{i=1..m}$ *along* P *and an* m*-walk* $\{\mathcal{Q}\}_{i=1..m}$ *along* Q *for some* m *such that for* $1 \leq i \leq m$*, either* $|\mathcal{P}_i| = 1$ *or* $|\mathcal{Q}_i| = 1$.

Intuitively, an m-walk basically partitions a chain into m non-empty subchains which preserves the order of points. Refer to Figure 2.11 for an example of an m-walk. On the other hand a paired walk consists of two partitions for a given m on two polygonal chains. The paired walk is only valid if for each $1 \leq i \leq m$ the i^{th} partition of at least one of the m-walks contains exactly one point.

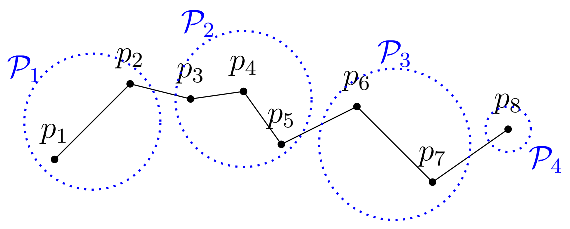

Figure 2.11: In the figure above, P is a polygonal chain with 8 vertices, an m-walk on P where $m = 4$ is shown above in 4 partitions. Notice all the points are covered and no partition is empty. Also no two partitions share any points.

An m-walk does not have any restrictions on the number of vertices of each partition. On the other hand, a paired walk on two polygonal chains P and Q has very restrictive rules. Notice that m must be at most the length of the shorter chain. Furthermore, for the i^{th} partitions $\mathcal{P}_i$ or $\mathcal{Q}_i$ must have exactly one vertex.

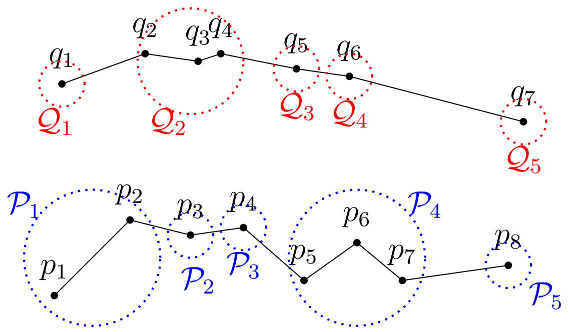

Figure 2.12: An example of a paired walk $(\mathcal{P}, \mathcal{Q})$. Observe that for any i, either $\mathcal{P}_i = 1$ or $\mathcal{Q}_i = 1$

Consider the dog and person scenario. Let P be the polygonal chain on which the person walks on and Q be the chain on which the dog walks on. At any point during the walk, there are three different cases:
(1) $|\mathcal{P}_i| > |\mathcal{Q}_i| = 1$: Person moves forward and dog stays.
(2) $|\mathcal{P}_i| < |\mathcal{Q}_i| = 1$: Dog moves forward and person stays.
(3) $|\mathcal{P}_i| = |\mathcal{Q}_i| = 1$: Both person and dog move forward.

Observe that a paired walk is the same as a coupling (Definition 2.7). A coupling is the same as a paired walk but the they are formulated differently. This new formulation will be helpful in designing an algorithm for simplification closed under Fréchet distance.

Definition 2.13 (Cost of a paired walk). T*he cost of a paired walk $W = (\mathcal{P}_i, \mathcal{Q}_i)$ along two paths P and Q is:*

$$d_F^W(P, Q) = \max_i \max_{(p,q) \in \mathcal{P}_i \times \mathcal{Q}_i} d(p, q)$$

The cost of a paired walk is defined for any valid paired walk on two polygonal chains. Recall the relationship between a paired walk and a coupling distance. Similarly, the cost of a paired walk is related to the length of a coupling (Definition 2.8). To be more precise, the cost of a paired walk is the same as the length of a coupling. For any possible coupling of two chains there is a paired walk. The length of the coupling is the same as the cost of the associate paired walk.

Definition 2.14 (discrete Fréchet distance). T*he discrete Fréchet distance between two polygonal chains P and Q is*

$$d_F(P, Q) = \min_W d_F^W(P, Q).$$

Consider all possible paired walks of two chains P and Q. Each paired walk has a specific cost. The paired walk with the smallest cost is called the Fréchet alignment of P and Q. Moreover, the cost of this paired walk is the discrete Fréchet distance. Observe the discrete Fréchet distance is the same as coupling distance(Definition 2.9).

It is clear that m-walk introduces a new formulation and terminology to describe coupling distance. For each definition above, the relationship between the m-walk terminology and the coupling terminology is given. Table 2.1 summarizes all of these relationships.

This new observation enables us to think of the coupling distance slightly at a different angle. Consider an m-walk on polygonal chain P. Each partition in this m-walk is associated with a disk. This disk is the smallest enclosing disk that contains all the points in the corresponding partition. Now consider a positive approximation value δ. All the disks must

Coupling Terminology	**m-walk Terminology**
coupling	paired-walk
coupling length	cost of paired-walk
coupling distance	discrete Fréchet distance

Table 2.1: Same mathematical elements with different formulation and terminology. This is a summary of these relationships. The Discrete Fréchet distance is just a different formulation of the coupling distance.

have a radius less than or equal to δ. Observe that such partitioning always exists. Each vertex can be partitioned separately. Thus, each disk will have a radius of 0 which is less than any positive value associated with δ. Consider the polygonal chain P' with the centres of the disks as its vertices in the order of the partitions. By the definition of discrete Fréchet distance, the distance between P and P' is at most δ. Imagine a paired walk on P and P'. The m-walk on P remains the same and each vertex of P' is partitioned separately. Now for each partition i in the paired walk, the m-walk on P' has exactly one point. This point is at most δ distance away from all the points in the i^{th} partition of the m-walk on P. Therefore, the discrete Fréchet distance between the P and P' is at most δ.

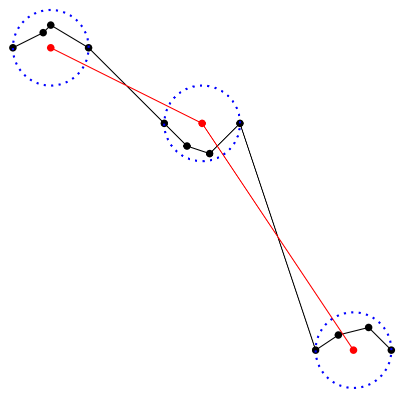

Figure 2.13: The black polygonal chain is the input chain. Each dotted blue disk corresponds to a partition which covers the points inside the disk. The red chain is the simplified version. This chain in obtained by connecting the centres of the disks in order of the partitions.

This observation enables us to design a greedy algorithm to approximate a polygonal chain P. Given a polygonal chain P with n vertices and a positive approximation value δ. Find an index i such that the first i elements of P can fit in a disk of radius δ but $i+1$ first elements cannot fit in any such disk. Append the centre of the disk to P' and recursively perform the same algorithm for $P[i+1..n]$. The only question remains to be answered is how to find the index i? Chapter 5 explains the details of this algorithm as it covers a very similar approach.

Chapter 3

Trajectories in surveillance systems

Chapter 2 reviewed polygonal chains and covered them formally. Trajectories are a special form of polygonal chain. In other words, a trajectory is the same as a polygonal chain with extra information. Each node in a trajectory stores not only the coordinate position of the node but also a timestamp. Trajectories are normally formulated when an object is tracked and the speed of the object is important. This chapter covers trajectories and defines them formally in general terms. It is also worth investigating how to define trajectories for our specific problem. Consider a vehicle being tracked by a surveillance system equipped with a camera. The underlying tracking system reads images from the camera and spots the vehicle once every frame. The camera provides the timestamp at which the frame was taken. The tracking system can store the size of the vehicle and maybe the colour histogram of the vehicle. More importantly, it can store the centroid of the vehicle and the timestamp of the corresponding frame.

3.1 Trajectory Definition

This section formally defines trajectories and illustrates how they can be captured and stored. In order to define trajectories formally, we need to first start by the definition of a spatio-temporal node.

Definition 3.1 (Spatio-temporal Node). A *spatio-temporal node is defined as* $c = (p, t)$ *where* $p \in \mathbb{R}^d$ *defines a point in a* d*-dimensional space and* $t \in \mathbb{R}_{\geq 0}$ *defines a timestamp in an arbitrary unit of time.*

Notice that t in Definition 3.1 is a positive real number. This value can be defined in any arbitrary unit of time such as seconds or the number of clock cycles in a CPU. This value

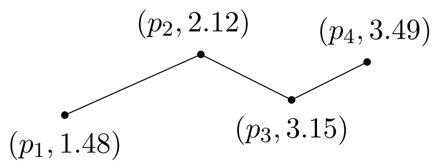

Figure 3.1: This figure illustrate an example of a trajectory. Notice the timestamp value increases at each spatio-temporal node compared to its previous node.

represents the amount of time passed from a specific time. At the end of this chapter, we will see that this specific time can be safely ignored in our problem. Thus, it is not much of a concern here in our definition.

Definition 3.2 (Trajectory). A *Trajectory* $T = (c_1, c_2, ..., c_n)$ *is an ordered sequence of spatio-temporal nodes such that for any* $0 < i < j \leq n$*, the timestamp* t_i *corresponding to* c_i *is restrictedly less than* t_j *corresponding to* c_j*. Furthermore, for any* $0 < i < n$*, the spatio-temporal node* c_i *is connected to* c_{i+1} *by an edge.*

3.2 Trajectory speed measurement

The speed of a moving object is very important when it comes to comparing trajectories. So far, we have a mathematical definition of a trajectory. In order to compare two trajectories, we need to assume a few things. The definition of a trajectory provides information about the position of the object at discrete points of time. However, objects move through space continuously. Figure 3.1 illustrates a trajectory that starts at time 1.48 and ends at 3.49. Given this trajectory, we also know the position of the object at times 2.12 and 3.15. A natural question that arises here is where was the object at time 2.0. Recall the person and dog scenario. Let us assume that one trajectory P is given for the person and another trajectory Q is given for the dog. Let us also assume that the dog and the person moved very closely relative to each other. However, Q only contains two spatio-temporal nodes and P contains 1000. When comparing P and Q, it would be useful to know where the dog was at times given by P and vice versa. This brings us to introduce some assumptions about the edges connecting a trajectory nodes.

One can introduce different models about the speed at which the object moves in between two adjacent nodes. One possible scenario can be a Gaussian-based random function. However, this can make the data inconsistent and therefore not a good model to follow. Another solution may be to assume that the object was stationary at the first vertex. Then as it moves toward the second node, the object accelerate by a constant factor. This constant factor can be calculated based on the timestamps of the two nodes at the end-segments of the edge. Once the object is on the second node, it has a certain speed. This model can be applied to all the other edges. The only difference is that the object has a certain speed calculated in the previous steps. Given such a model, one can compute where the object was at any point during its movement. In order to find the position of the object for a given time, two parameters are required: acceleration and initial speed. The acceleration is the acceleration of the object on a given edge. The initial speed is the speed of the object at the end segment of an edge with the lower timestamp. The speed of the object at the other end segment is required to be calculated in order to compute for the acceleration of an edge:

$$a = \frac{s_2 - s_1}{t_2 - t_1} \tag{3.1}$$

In the equation above, a is the acceleration of an edge, s_1 and s_2 are the speeds of the object at the end segments of an edge and t_1 and t_2 are the timestamps associated with each end segment. When a trajectory is given as input, the assumption was that the speed at the first vertex is zero. This assumption allows us to have a starting point but the acceleration of the first edge needs to be calculated. (3.1) is not enough to compute the acceleration because s_2 is unknown. However, the length of the edge can be computed in constant time. Given the distance the object moves and the two timestamps and the initial speed of the object, s_2 can be calculated. Let us first consider the following equation:

$$d = s_1(t_2 - t_1) + \frac{1}{2}a(t_2 - t_1)^2 \tag{3.2}$$

$$d = s_1(t_2 - t_1) + \frac{1}{2}(\frac{s_2 - s_1}{t_2 - t_1})(t_2 - t_1)^2 \tag{3.3}$$

Let Δt denote for $t_2 - t_1$. Let us simplify the expression above:

$$d = s_1\Delta t + \frac{1}{2}(s_2 - s_1)\Delta t \tag{3.4}$$

$$d = s_1\Delta t + \frac{1}{2}\Delta t s_2 - \frac{1}{2}\Delta t s_1 \tag{3.5}$$

$$d = \frac{1}{2}\Delta t s_2 + \frac{1}{2}\Delta t s_1 \tag{3.6}$$

In the next step, equation (3.6) is used to solve for s_2.

$$s_2 = \frac{2d}{\Delta t} - s_1 \tag{3.7}$$

Consider the following problem. Given a time at which the object was travelling on the last edge of a trajectory excluding the end segments of the edge. Find the position of the object on this edge. In order to find the position of the object, the acceleration of the edge needs to be calculated. This leads us to conclude that we need to compute the speed of the object at both end segments of the edge. In order to do this, we need to go back to the previous edge and so on. Finding the position of the object based on a given time has a linear solution and this is undesirable. A quick solution to the problem is to pre-compute the speed at each node and keep them in memory. Although this may seem to be a reasonable solution, this may not be the best solution to our specific problem. The clustering system may be dealing with a large number of trajectories. Pre-computing the speed for each vertex may affect the performance of our system. We seek for a simpler reasonable solution.

In general, objects detected by the object tracking system may move at a constant speed. Moreover, these objects move in a 3D space and assuming a 2D acceleration vector may not be appropriate. The other solution is to assume that the objects move at a constant speed on each edge. It is trivial to compute this constant speed on any given edge:

$$s = \frac{d}{\Delta t} \tag{3.8}$$

Equation (3.8) can be used to compute the speed of the object on an edge but this may not be required. Recall, in the second solution, we had to compute the acceleration of an edge and the speed of the object. Without these parameters, we would be unable to determine the position of an object on an edge. Consider an edge e with two spatio-temporal end points c_i and c_{i+1}. Let p_i, p_{i+1} and t_i, t_{i+1} denote the points and times associated with these end points. We consider the following: Given a time t_x such that $t_i < t_x < t_{i+1}$ and we would like to determine the position of the object on edge e at time t_x. Since the assumption is that the object is moving at a constant speed, we can compute what fraction of the length of the edge must be passed at time t_x:

$$\frac{t_x - t_i}{t_{i+1} - t_i} \tag{3.9}$$

This fraction can be used to find the position of the object p_x at time t_x:

$$p_x = (\frac{t_x - t_i}{t_{i+1} - t_i})(p_{i+1} - p_i) + p_i \tag{3.10}$$

3.3 Last spatio-temporal node assumption

So far, we have made one major assumption about trajectories and that is that there is no acceleration on the edges. In other words, objects move at a constant speed on these edges.

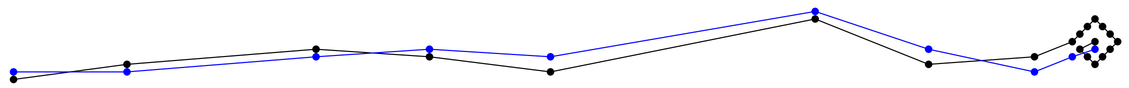

Figure 3.2: The two trajectories (black and blue) look very similar but the black trajectory is longer. A good trajectory distance function should find these two trajectories very similar.

Imagine comparing two trajectories to each other of different time interval. One trajectory may be a minute long and the other one may be one hour long. In a surveillance system, the last node of a trajectory may occur in two ways. The target object has stopped moving and therefore its trajectory only provides information for the time that the object was moving. The second case may be that the object moved out of the surveillance view which leaves no way to further capture its movements. Unfortunately, in the latter case, we can only work with the information we have available to us. Let us focus on the first case. Imagine the following scenario.

Consider a person ”A” walking on a path and then at some point the person reaches a bench. The person then goes around the bench a few times and then decides to sit on the bench. A second person ”B” takes a very similar path at almost the same speed as person ”A” but as soon as the person ”B” reaches the bench, the person sits on the bench. The tracking system tracks objects when they are moving and it stops tracking them when they are stationary. In this case, person ”A” is tracked on a very similar path and then around the bench a few times. Furthermore, once the person sits down, the tracking system stops and generates a trajectory T_A. Person ”B” is similarly tracked but since the person sits on the bench right away, the trajectory T_B is shorter. The question is how to compare these two trajectories. Notice that both persons in this case took a very similar path but the trajectories generated by the surveillance system may differ substantially in length.

In order to solve this issue, we can assume that for any time after the last timestamp, the position of the object is the position of the last node. In other words, we assume that once the surveillance system stops tracking, the object remained stationary. In practice, this happens frequently in our surveillance system. For example, cars drive into the surveillance view and park. The surveillance system tracks them as they enter the scene until they park.

3.4 Frame-based Time Units

Definition 3.2 does not require a specific unit of time for the timestamp representation. Moreover, as long as the time unit is consistent across all timestamps, all of the mathematical definitions remain valid. In a surveillance system, a camera captures images at a constant rate. In other words, the time gap between any two consecutive frames is a constant. This enables us to define our time unit in terms of the frame rate. In other words, we can define our time unit to be the time between any two consecutive frames. For example, if the surveillance camera captures images at a rate of 15 frames a second then the time unit will be $\frac{1}{15}$ second. As long as the frame rate remains the same, the clustering system should remain consistent. This allows us to assume that for a given trajectory from the tracking system, the timestamps increment by one unit for each subsequent node.

3.5 Time-Independent Trajectory

When clustering trajectories, the time at which each trajectory occurred is not important. The speed of the object moving at each node is important which can be represented using the timestamps. If two trajectories occurred two days apart at very similar speed taking very similar paths, they must be clustered in the same group. In other words, it is not important when the first node of a trajectory was recorded. Due to this fact, we can assume that the timestamp of the first node of any trajectory is zero. The timestamp increments relative to the previous node at each step. Our clustering system will take advantage of this fact.

Definition 3.3 (Time-independent Trajectory). C*onsider a trajectory* $T = \{c_1, c_2, ..., c_n\}$ *where* t_i *is the timestamp associated with* c_i*. Trajectory* T *is called a time-independent trajectory if and only if* $t_1 = 0$.

3.6 Conclusion

We have learned about spatio-temporal nodes and representation of a trajectory. Moreover, we have also seen how to use timestamps to measure the velocity of a trajectory at different positions along the trajectory path. Furthermore, this chapter investigated the discrete representation of trajectories while conceptually treating them as continuous elements. This representation is very powerful because it formally defines the position of the coordinates on a trajectory while not explicitly defined. This enables us to uncover some hidden information about a trajectory when it is being compared against other trajectories. As we will see, this will be used extensively when computing the distance of two trajectories. The other useful property of this representation is that these coordinates can be computed very efficiently with the means of an algorithm.

Although this representation is very powerful, it is not sufficient to remove all the ambiguities about the coordinates of a trajectory. We needed one extra assumption and that was about the position of the object beyond the timestamp of its last node. With this last assumption, the position of trajectory is defined for any valid timestamp from zero to infinity. This provides a solid platform to compare trajectories. All of these points will be used to construct a distance function without any ambiguities that in turn can be used to compare trajectories.

Chapter 4

Trajectory Distance Function

In Chapter 3, a formal definition for a trajectory is given. Furthermore, Chapter 3 covers time-independent trajectories. However, the definition of a trajectory is not sufficient to compare two trajectories. We need to define a distance function that preserves the similarities of trajectories. It should also be able to distinguish the differences of two trajectories. This chapter covers the distance function used for the surveillance system. Moreover, Section 4.3 shows that this distance function is a metric distance function. Finally, this chapter covers the workings of an algorithm that can compute the distance function in a very fast fashion.

4.1 Requirements of a Good Distance Measure

It is useful to investigate the properties of a good distance function and determine what types of requirements it has to meet. This will help us in the process of searching for a good distance function.

Let us first start with a generic solution. In other words, let us consider the properties of a generic distance function rather than a distance function specifically to compare trajectories. The distance measure must be a value that represents the differences between two objects. Consider the set of all real numbers R. Furthermore, let us define two values in this set: $\{a, b\} \subset \mathrm{R}$. Let us try to address the following problem: what is a good distance function $d(a, b)$ that can capture the differences between a and b. For example, is the following function a good candidate for our solution:

$$d(a, b) = a - b \tag{4.1}$$

The first property that comes to mind is that the distance function is supposed to measure the ”distance” between a and b. Hence, the order in which a and b are given to us should not matter. In other words, $d(a, b)$ must evaluate to the same value as $d(b, a)$. This is called the symmetric property of a function and it is a very natural solution when it comes to distance functions. The distance function must have this property to ensure that the distance between two objects are captured regardless of the order they are given to the function. The distance function defined above does not meet this requirement and therefore it is not a good solution.

The second property of a good distance function is that it has to have a starting point. We need to have a base value for identical objects and as objects become more and more dissimilar, the value of the distance function should grow. For example, in the case above, if a and b are completely identical, then the distance value must be zero reflecting the fact that they are identical. On the other hand, if a remains constant and b grows on the positive direction then their distance must also grow reflecting how much b has distant itself from a. Moreover, if b moves in the positive direction and also a moves in the negative direction then their differences are now even larger. For example, consider the function distance function:

$$d(a, b) = |a - b| \tag{4.2}$$

This distance function preserves both of the properties we have defined by now: First of all, the following assignment is always true: $d(a, b) = d(b, a)$. Secondly, if a and b are equal then, $d(a, b)$ evaluates to zero meaning that a and b are identical. Having a base value is important to reflect what objects are identical and it also provides a way of comparing the actual distance values themselves. For example, with the new distance function, $d(2, 4)$ evaluates to 2 but $d(2, 1002)$ evaluates to 1000. This captures the fact that 4 is much more similar to 2 rather than 1002.

Another important requirement of a distance function is that it has preserve its meaning. Consider three points p_1, p_2 and p_3. Consider someone walking from point p_1 to p_2 and then to point p_3. The total ”distance” that the person walks is at least equal to the distance if the person walked directly from p_1 to p_3. This is a very important property both algorithmically and conceptually.

We have covered some generic requirements of a good distance function. Let us now focus specifically on trajectories and determine what is a good distance function for a trajectory. Trajectories are ”paths” that have extra information. Paths are mathematical curves in multi-dimensional spaces. They have a starting point and also an ending point. Two curves are considered to be very similar if they start around the same point in space and end around the same point and also take almost the same path. If two curves start at very distant points then immediately they cannot take similar paths and therefore they will be very different from one another. The same analogy applies to the case where two curves end at two very distant points. This is what Fréchet distance captures. It compares two curves from their starting points and follows them to their final points. It also computes their distance as the two points move through space to get to their final positions. The Fréchet distance also satisfies all the generic requirements of a good distance function. The only problem is that it is designed for curves and not for specific curves like trajectories. Trajectories have paths but they also move at different speeds along their paths. Capturing the differences of their motion and velocity is also important. We will try to use Fréchet distance as a starting point and define a new distance function that can also capture the velocity differences of the objects.

4.2 Trajectory Distance Function Definition

Chapter 2 gave an overview of Fréchet distance function. This function is a similarity function for polygonal chains. Recall that polygonal chains have no information about time. They are basically an ordered sequence of points in a d dimensional space. The Fréchet distance attempts to measure the distance between two chains by imagining a speed associated with each chain at any point. Intuitively, the Fréchet distance assumes that the two chains are the paths of two objects taken at the same time interval. Moreover, the Fréchet distance considers the most optimal speeds at various points on each chain such that the distance between any two points at any time is minimal. In a sense, this distance function is a good starting point for us. That is due to the fact that it attempts to preserve the similarities of the paths. However, trajectories are mathematical objects with more information associated with them. The time intervals between any two nodes can be computed and also the speed of the moving object can be calculated. Moreover, we would like our distance function not only to capture the curve dissimilarities but also their velocity at different points. Let us step back and look at the Fréchet distance more closely.

Definition 2.6 includes two reparameterization mapping function α and β. These two functions basically map a time t into a point on the polygonal chains P and Q respectively. The aim of the Fréchet distance is to find the optimal reparameterization mappings. However, notice that these two functions are provided to us when dealing with trajectories. Recall, Chapter 3 reviewed the speed of moving objects along the edges of a trajectory. Our solution was to assume that the object moves with a constant speed on the edges. Moreover, we showed how the location of an object for some input time can be computed using the equation (3.10). This equation can help in replacing α and β in Definition 2.6.

Equation (3.10) is defined on the edge of an input trajectory. Consider a trajectory T with

$t_1, t_2, ..., t_n$ as its timestamps. Equation (3.10) is not defined for $t > t_n$. We need to introduce a trajectory reparameterization mapping that maps any time $t \geq 0$ to a point p on a trajectory.

Definition 4.1 (Trajectory Time Function). L*et* $T = \{c_1, c_2, ..., c_n\}$ *be an arbitrary trajectory. For an integer* i *such that* $0 < i < n$*, let* t_i, p_i *be the timestamp and the point associated with* c_i *respectively. Furthermore, for some value of* $t \geq 0$*, let* j *be the largest integer such that* $t \geq t_j$*. The trajectory time function is defined as:*

$$\alpha_T(t) = \begin{cases} (\frac{t-t_j}{t_{j+1}-t_j})(p_{j+1} - p_j) + p_j & \textit{if } j < n \\ p_n & \textit{if } j = n \end{cases}$$

The cost of computing the trajectory time function can be a disadvantage. Assuming that the trajectory nodes are stored in a linked list, the computation time will be linear. Moreover, if nodes of a trajectory are stored in an array then j can be found using a binary search. Therefore, the best we could do in this case is $O(\log n)$. However, this is only true when a single trajectory is given with a timestamp $t \geq 0$ and the problem is to find the point associated with t. The good news is that we do not use the trajectory time function in such a way. This function will only be used when computing the distance between two trajectories. We will come back to this point after explaining the distance function.

4.3 Metric Distance Function

So far, we have explored the relationship between points in space and timestamps of a single trajectory. These details will be used to compute the distance between two given trajectories. Recall our example of a person walking a dog. Both the dog and the person start at the timestamp 0 at two different positions. As the dog and the person walk along their path, the distance between them may fluctuate at different timestamps. Furthermore, the dog or the person may stop walking at different timestamps. In other words, their trajectories may be defined through different spans of time. For example the person may walk for two minutes and stop while the dog may walk for three minutes. As a result the trajectory associated with the dog will be longer in time. This is not an issue because the position of the person is still defined after the second minute as we discussed in the previous chapter. Let us now define the time-distance function.

Definition 4.2 (Time-Distance Function). L*et* P *and* Q *be two different trajectories. The time-distance function* $d_{PQ}(t)$ *is defined for all values of* $t \geq 0$*:*

$$d_{PQ}(t) = d(\alpha_P(t), \alpha_Q(t)) \quad ,$$

where α_P, α_Q are the trajectory time functions of P and Q respectively. Furthermore $d(p, q)$ is the euclidean distance between the points p and q.

Notice that the time-distance function is undefined for any negative values of t. Moreover, observe that the time-distance function evaluates to a constant value for values of t not in the range of timestamps defined in P and Q. In our example above, the time-distance function evaluates to a constant value for any value of $t \geq 3mins$. Another important property of time-distance function is that it is symmetric. In other words, the time-distance function is symmetric: $d_{PQ}(t) = d_{QP}(t)$. This is a crucial point because as it will be clear by the end of this chapter, the distance function is symmetrical as a result of this property.

The proof of this property is very intuitive: Given two trajectories P and Q and their trajectory time functions $\alpha_P(t)$ and $\alpha_Q(t)$. The claim is $d_{PQ}(t) = d_{QP}(t)$. Furthermore, we have $d_{PQ}(t) = d(\alpha_P(t), \alpha_Q(t))$ and $d_{QP}(t) = d(\alpha_Q(t), \alpha_P(t))$. Since the euclidean distance function is symmetrical $d(p, q) = d(q, p)$, therefore $d_{PQ}(t) = d_{QP}(t)$.

The time-distance function is the basis of our trajectory distance function. As opposed to the Fréchet distance function, we have more information to utilize. In the Fréchet distance method, the speed of the two curves at different points of time is undefined. As a result, the Fréchet distance method simply assumes that the dog and the person move at speeds such that they stay as close to each other as possible. This is in fact a good property for the Fréchet distance because it attempts to evaluate the similarities of two polygonal curves. If two curves are drawn on a piece of paper, the speed at which they are drawn on the page is meaningless when they are being compared to one another. In other words, the speed at which the two curves are drawn do not reflect the similarities of the two curves. This is not the case when the task is to compare trajectories. In fact, when it comes to trajectories, the speed of the object moving through space is very crucial. For example, a person and a vehicle can move on the same path at very different speeds. When comparing their trajectories, a good deal of dissimilarity must be represented. Our trajectory distance function is basically the same as the Fréchet distance but with a very important difference. The trajectory distance function considers the actual speeds at which the objects move through space because these details are available to us. Let us go back to the example with the dog and the person and assume that we had the trajectories of the dog and the person rather than their polygonal chains. Intuitively, the trajectory distance function occurs when the dog and the person are the farthest away from each other. This distance is the trajectory distance and it has very important properties that we will exploit to cluster our trajectories. Let us first formally define the trajectory distance function and then explore its main properties.

Definition 4.3 (Trajectory Distance Function). L*et P and Q be two different trajectories. The trajectory distance function $\delta_T(P,Q)$ is defined as following:*

$$\delta_T(P,Q) = \max_{t\geq 0}(d_{PQ}(t)) \quad ,$$

where $d_{PQ}(t)$ is the Time-Distance function of P and Q at time t.

Let us first make a very important observation. The trajectory distance function defined above will always evaluate to zero or a positive real value. It is clear why the value can never be negative. It is also important to note that the value can never be infinite. Although the function is defined as the maximum value of an open-ended set, one can observe this at a slightly different angle. There exists a value t_{max} such that for all the values greater or equal to t_{max}, the function d_{PQ} evaluates to a fixed value. This value is the larger timestamp of the final nodes of the trajectories. As a result we can think of the trajectory distance function this way:

$$\delta_T(P,Q) = \max_{0\leq t\leq t_{max}}(d_{PQ}(t)) \tag{4.3}$$

Since this is a closed set, the function must evaluate to a non-negative finite real number. This is a very important observation as it is required by metric distance function that we will investigate next.

There is a large number of clustering algorithms in the literature. However, our requirements limit us to use a certain number of them. In particular, we would like our clusters to evolve as we collect trajectory data. Our goal is to have a tracking system in real time where we maintain our clusters as we detect trajectories in real time. This means that the trajectory dataset is not available as a whole for the clustering algorithm to use. There are numerous clustering algorithms out there that use an entire given set before starting their task. This is not the case for us. We need our clusters to be maintained incrementally. That is, as we discover trajectories, we would like to insert it into the right cluster. This is a challenging task because clusters can evolve and there might be elements in different clusters that can be exchanged to produce better clustering groups. The other problem is that trajectories may take a lot of space and keeping them all in memory may not be a viable option. This means that some of the trajectories may have to be transfered into the hard drive as clusters are evolving. These trajectories may find their way back into main memory as other trajectories may be transferred to the hard drive. These issues may take away some generic solutions from us when it comes to choosing the right clustering algorithm. Furthermore, as the number of candidate clustering algorithms decreases because of these constraints, our distance function must be more general and broad. Having a general distance function can help us freely choose the right clustering algorithm for our problem.

Let us now explore some broad requirements that are imposed by most clustering algorithms and ensure that our clustering algorithm satisfies those requirements. General clustering algorithms accept well-defined objects as input and try to group them by their similarities. For them to achieve this, they need a tool to measure the similarities and dissimilarities of the input objects. This tool is generally a distance function which accepts two objects as input and returns a real value expressing the dissimilarities of the two objects. Normally, distance functions are formulated such that a bigger value is the indication of dissimilarities between the two objects while a smaller value indicates their similarities. We have already defined our trajectory distance function in Definition 4.3 which is capable of comparing two trajectories and reflect their similarities. We have also discussed why this measurement is a good tool for comparing trajectories. Let us now discuss some broad requirements imposed on such distance functions. One of the mostly imposed requirements of a distance function is that it needs to be a metric distance.

Recall Definition 2.2 that defined metric space ordered pairs. This definition consists of two parts. M is a set of objects and d is a distance function between any pair of objects in M. For a distance function to be metric, it needs to meet four requirements listed in that definition.

The first requirement is that $d(x, y) \geq 0$. The trajectory distance function is the maximum of a non-negative function. This simply confirms that the distance function will never evaluate to a negative value. The first requirement is therefore met.

The second requirement is $d(x, y) = 0$ is only the case if and only if x and y are equal. Let us start by assuming that we have two trajectories P and Q such that $\delta_T(P, Q) = 0$. This implies that for all values of $t \geq 0$, the position of the object associated with trajectory P and Q are the same. This confirms that the two trajectories P and Q are equal to each other. This covers the first part of the proof, namely if $\delta_T(P, Q) = 0$ then $P = Q$. Let us now assume that $P = Q$. This means that for any $t \geq 0$, the position of the objects P and Q are the same. This implies that for any value of $t \geq 0$, the euclidean distance between the two points associated to P and Q at time t is equal to zero. Moreover, this means that the trajectory distance function between P and Q is equal to zero.

The third requirement is the symmetric property of the function. For any two trajectories P and Q, $\delta_T(P, Q)$ must be equal to $\delta_T(Q, P)$. Earlier in this chapter, we showed that the two time-distance functions $d_{PQ}(t)$ and $d_{QP}(t)$ are always equal. Let t be the value of the timestamp at which P and Q are furthest away from each other. The trajectory distance function $\delta_T(P, Q)$ evaluates to $\max\limits_{t \geq 0} d_{PQ}(t)$. On the other hand, the distance function $\delta_T(Q, P)$ evaluates to $\max\limits_{t \geq 0} d_{QP}(t)$. Since $d_{PQ}(t) = d_{QP}(t)$, therefore $\delta_T(P, Q) = \delta_T(Q, P)$.

The last property is the triangulation property of the distance function. This one involves a more detailed proof. Let A, B and C be three different trajectories. Let AB_t denote the distance between the two curves A and B at time t. Furthermore, let t_{AB} be the timestamp at which the distance between A and B is maximum. Observe that $AB_{t_{AB}}$ is the trajectory distance between A and B:

$$\delta_T(A, B) = AB_{t_{AB}} \tag{4.4}$$

$$\delta_T(B, C) = BC_{t_{BC}} \tag{4.5}$$

$$\delta_T(A, C) = AC_{t_{AC}} \tag{4.6}$$

Consider objects A', B' and C' be associated with curves A, B and C respectively. Now imagine as A', B' and C' are moving through space, we take a snapshot of their position at time t_{AB}. This provides us with three points in space. Imagine these points as the corners of a triangle. Observe that the length of the sides of this triangle are equal to $BC_{t_{AB}}$, $AC_{t_{AB}}$ and $AB_{t_{AB}}$. We obtain the following from the triangle inequality law:

$$BC_{t_{AB}} + AC_{t_{AB}} \geq AB_{t_{AB}} \tag{4.7}$$

Furthermore, the following two inequalities hold because the right hand side of each inequality is the trajectory distance of the corresponding two trajectories:

$$BC_{t_{AB}} \leq BC_{t_{BC}} \tag{4.8}$$

$$AC_{t_{AB}} \leq AC_{t_{AC}} \tag{4.9}$$

Summing up both sides of the two inequalities, we get:

$$BC_{t_{AB}} + AC_{t_{AB}} \leq BC_{t_{BC}} + AC_{t_{AC}} \tag{4.10}$$

We take the argument we derived above and apply what we observed in equation 4.7:

$$AB_{t_{AB}} \leq BC_{t_{AB}} + AC_{t_{AB}} \leq BC_{t_{BC}} + AC_{t_{AC}} \tag{4.11}$$

This simply shows the following:

$$\delta_T(A, B) \leq \delta_T(B, C) + \delta_T(A, C) \tag{4.12}$$

This proves the triangulation property of our trajectory distance function. We proved all four requirements that a metric distance function needs to satisfy and therefore the trajectory distance function is a metric distance function.

4.4 Discrete Distance Function Optimality

This chapter has covered the definition of the trajectory distance function that is capable of comparing two input trajectories. However, it is completely unclear how to design an algorithm that can compute this distance. In this section, we show that there is a discrete version of this distance function. It turns out that the discrete distance function is the same as the trajectory distance function that we have already defined. The discrete trajectory distance function is the basis of our algorithm. The idea of the discrete trajectory distance is to use only the vertices and their timestamps to compute the distance.

Definition 4.4 (Discrete Trajectory Distance Function). L*et $P = \{p_1, p_2, ..., p_n\}$ and $Q = \{q_1, q_2, ..., q_m\}$ be two given trajectories. Furthermore, let tp_i denote the timestamp corresponding to the node p_i and tq_i be the timestamp corresponding to q_i. The discrete trajectory distance function $\delta_{dT}(P, Q)$ is defined as following:*

$$\delta_{dT}(P, Q) = \max(\max_{1 \le i \le n}(d_{PQ}(tp_i)), \max_{1 \le i \le m}(d_{PQ}(tq_i)))$$

where $d_{PQ}(t)$ is the Time-Distance function of P and Q at time t.

Although the definition of the discrete version looks very different compared to the original trajectory distance function, the two are in fact exactly the same. In the discrete version, we are only concerned with the timestamps at the vertices of the two trajectories. However, in the original version, all the timestamps that are greater or equal to zero are taken into account. The question is why this is a good approach and the answer is simple: the two functions are basically equal and they evaluate to the same value. However, it is very difficult to implement an efficient algorithm for the original function but the discrete version has a simple algorithmic solution.

Let P_t denote the point associated with the position of a trajectory P at time t. Moreover, let P and Q be two arbitrary trajectories. Furthermore, let t be the timestamp at which P and Q are furthest away from each other. It is clear that the euclidean distance between P_t and Q_t is the trajectory distance between P and Q. The only way that the original trajectory distance function and the discrete trajectory distance function can evaluate to two different values is the following scenario: If neither of P_t and Q_t are on the vertices of P and Q respectively and they lie on two edges of the two trajectories. Let us denote these two edges with P_e and Q_e associated with trajectories P and Q respectively. Moreover, let $[t_1, t_2]$ be the time interval shared by these two edges. Consider a distance function $d(T) = \|P_T Q_T\|$ defined over this time interval. For our scenario to be true, $d(T)$ must have a local maxima $T = t$. For the rest of this section, we show that such a maxima cannot exist and therefore we prove that the trajectory distance function and the discrete trajectory distance function are completely equal.

Imagine two points moving through space on two different straight lines at two different constant speeds. Consider the distance between these two points as a function of time. As time passes, there are two fundamental different cases:

1) The two points move on the same line or on two parallel lines

2) The two points move on two nonparallel lines

The first fundamental case consists of three sub-cases. The first one is when the distance between the two points does not change. This is a simple case and it can only occur when the two points are moving in parallel to each other or on the same line and also in the same direction at the same speed. This is a very simple case because it is clear that the maximum distance is the distance between the two points at all time and there is no local maxima.

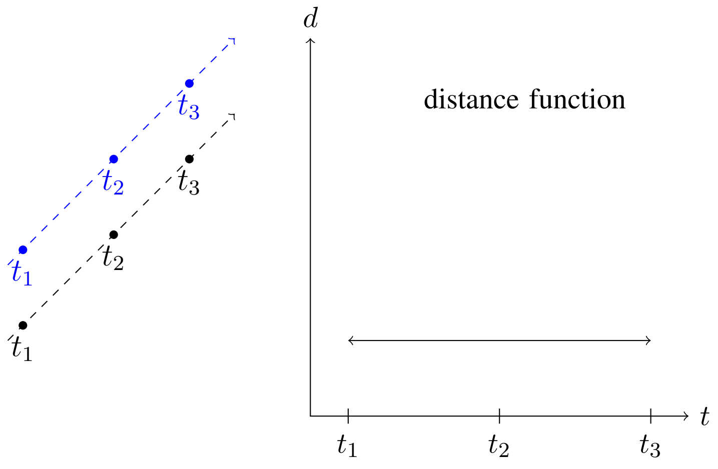

Figure 4.1: This figure illustrates the case where two points move in parallel lines at the same speed in the same direction. In the figure above on the left, two points are illustrated moving at the same speed on two parallel line. The distance function with respect to time is illustrated on the right hand side. Observe that this function does not have a global maxima. The distance between the two points is always constant.

The second sub-case occurs when two points move in opposite directions on the same line or on two parallel lines. In this case, the two points get closer to each and then at some point they move away from each other. In other words, the distance between them decreases until they reach a global minima and then they move away from each other. Once they reach their minimum distance, they start moving away from each in opposite direction and therefore their distance increases again. Observe that there is a global minima but there is no global maxima (Figure 4.2). Consider an interval of the distance function, the maxima occurs at one of the end points of the interval.

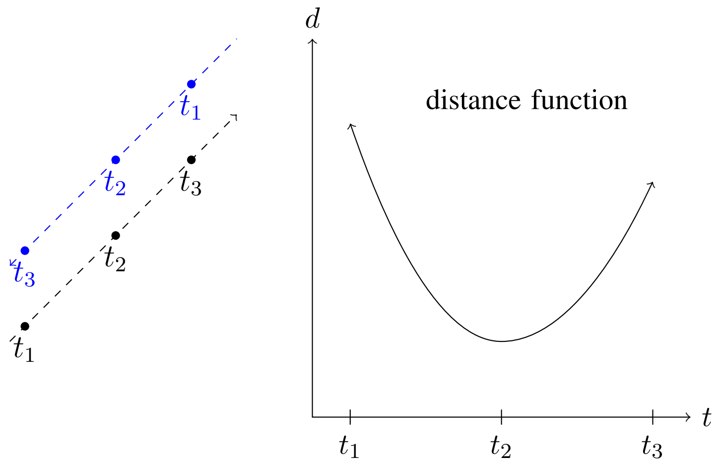

Figure 4.2: This figure illustrates the case where two points move in parallel lines in opposite directions. Note that speed of the points will not affect the results here. The speed at which the two points travel can affect the parabola but observe that it will not create a global maxima. The two points move in opposite directions and three snapshots of the two points is illustrated at times t_1, t_2 and t_3. The distance function with respect to time is illustrated on the right hand side. Observe no global maxima can ever occur.

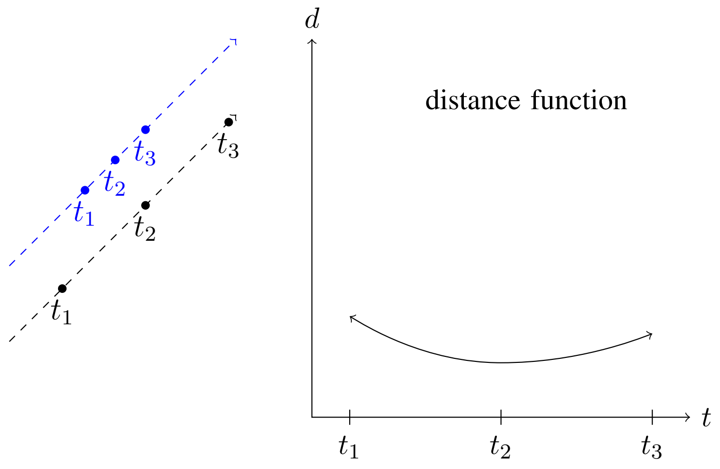

Figure 4.3: This figure illustrates the case where two points move in parallel lines in the same direction at different speeds. In this case, as the two points travel from time t_1 to time t_2, the distance between the two will be decreasing. The distance of the two objects is at a minimum at time t_2. Since one point is traveling faster than the other, its distance from the blue point increases after time t_2. Observe that no global maxima exists in the distance function again. There is a local minima when the two points get as close to each other as possible.

The third sub-case is when two points move on the same line or two parallel lines in the same direction but at varying speeds. This one is similar to the previous case. The two points may approach each other as one moves faster than the other one and therefore their distance decreases until they reach a global minima. Once they reach their minimum distance, the faster point passes the slower one and therefore their distance increases. Similar to the previous case, no global maxima can occur. There is only a global minima. If we look at an interval of the distance function, the maxima occurs at one of the end points of the interval.

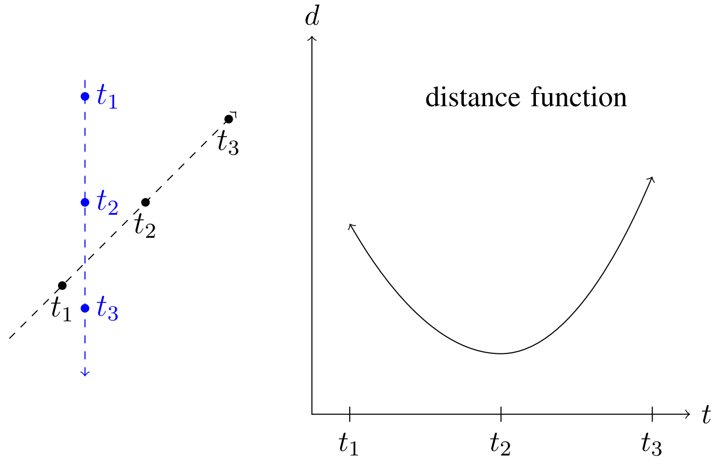

Figure 4.4: This figure illustrates the second fundamental case where the two lines are non-parallel. Considering the time interval of $(-\infty, +\infty)$, the two points will approach each other at a minimum distance (t_2). Furthermore, as t approaches $+\infty$, the distance between the two points increases again. There no global maxima. Only a global minima can occur.

The second fundamental case is when the two points are moving in two nonparallel lines. This case is easier to analyze. The most important thing to observe is that since the points are moving on two nonparallel lines, we can imagine the distance between them for the time interval $(-\infty, +\infty)$. Since the two lines are not parallel to each other, there is a point in time at which the distance between the two points is at a minimum. In other words, the distance at time $-\infty$ is $+\infty$ and as we move through time, the distance between the two points decreases until they reach a minimum distance and then as we move into time $+\infty$, the distance between the two increases again approaching $+\infty$ again. It is clear that there is no global maxima in this case again. There is only a local minima. Considering an interval of the distance function, the maxima can only occur in one of the two end points of the interval.

4.5 Distance Function Algorithm

In Section 4.4, we showed that the discrete distance function can be used to compute the trajectory distance between two trajectories. The discrete distance function is only concerned with the vertices of the input trajectories. This is a very important observation when we need to design an algorithm. In this section, we will start with a naive algorithm that can be used to compute the distance between two trajectories. This algorithm will only be used to illustrate the idea but it runs in $\theta(n.m)$ where n and m are the number of vertices in the two input trajectories. We will then present an efficient algorithm that does that computation and it runs in $\theta(n+m)$. This will be a very good improvement because it will guarantee fast distance computation which is the key to clustering large datasets.

Let us first start with the naive algorithm. In Section 4.4, we looked into the trajectory distance function more closely and we made an important observation. The observation was that the trajectory distance can only occur at the vertices of the trajectories. This implies that we can compute the trajectory distance by finding the distance between the two trajectories only at the timestamps of the vertices. Let us start by defining a simple function that computes the position of a trajectory at a given time $t \geq 0$. We also assume that each trajectory is stored in an ordered linked list.

Algorithm 1 Find the position of the trajectory at time t by doing a linear search

```
Require: $T$: input trajectory
Require: $t$: input timestamp
Ensure: Compute the position of $T$ at time $t$.
  procedure FINDPOSITIONBYLINEARSEARCH($T$, $t$)
      ASSERT($t \geq 0$)
      $n \leftarrow head(T)$
      while $next(n) \neq nil$ do
          $n' \leftarrow next(n)$
          if $t \geq timestamp(n)$ & $t \leq timestamp(n')$ then
              return $(\frac{t - timestamp(n)}{timestamp(n') - timestamp(n)})(position(n') - position(n)) + position(n)$
          end if
          $n \leftarrow n'$
      end while
      return $position(n)$
  end procedure
```

It is important to note that Algorithm 1 runs in linear time. The algorithm basically follows the edges of a trajectory in order. Each edge of the trajectory has a constant speed and if timestamp t occurs on that edge, the position associated with t can be computed. The algorithm ensures that t is a non-negative real number. If all the edges are inspected and t does not fall onto any of the edges, then t must be larger than the timestamp of the last vertex. Our assumption was that once the object reaches the last timestamp then it remains in that position for all timestamps larger than the last timestamp. Due to this assumption, the position of the last node will be returned in that case. The algorithm keeps track of the end points of the edge that is being inspected at each iteration of the while loop and names them n and n'. If t is between the timestamp of n and n', then its position is computed using equation (3.10). If the algorithm does not return a value while executing within the while loop, it will exit the while loop. At this point, n is the last node of the trajectory and therefore its timestamp is returned.

Algorithm 2 This algorithm takes two trajectories A and B as input. It then iterates over all the vertices of A and computes the maximum distance from the end points of A to their corresponding points on B. Algorithm 1 is used to find the corresponding point on B. Note that this function does not compute the trajectory distance between A and B because vertices of B are not inspected.

```
Require: $A$: first trajectory
Require: $B$: second trajectory
  procedure OneDirectionalDistanceComputation($A$, $B$)
      $n \leftarrow head(A)$
      $maxDist \leftarrow 0$
      while $n \neq nil$ do
          $p \leftarrow position(n)$
          $q \leftarrow$ FindPositionByLinearSearch($B$, $timestamp(n)$)
          $maxDist \leftarrow \max(maxDist, euclideanDistance(p, q))$
          $n \leftarrow next(n)$
      end while
      return $maxDist$
  end procedure
```

Algorithm 2 does not compute the trajectory distance between A and B. It does a one-directional distance computation *i.e.*, only the vertices of A are taken into account. In Section 4.4, we showed that the maximum distance between two trajectories can occur at the vertices of either trajectories. Thus, we need to look at all the timestamps at the end points of both A and B. Algorithm 2 is very intuitive: It iterates over the vertices of A and finds the corresponding points on B using Algorithm 1. The algorithm computes the distances at these timestamps and keeps track of the maximum distance in the variable $maxDist$. Once all the vertices are visited, the value of $maxDist$ is returned. In order to compute the actual trajectory distance, this computation must be done in both directions. This is simple:

Algorithm 3 This algorithm computes the trajectory distance between A and B by executing Algorithm 2 in both directions.

```
Require: $A$: first trajectory
Require: $B$: second trajectory
  procedure TwoDirectionalDistanceComputation($A$, $B$)
      $d_{AB} \leftarrow$ OneDirectionalDistanceComputation($A$, $B$)
      $d_{BA} \leftarrow$ OneDirectionalDistanceComputation($B$, $A$)
      return $\max(d_{AB}, d_{BA})$
  end procedure
```

This way all the vertices of both trajectories are taken into account. The algorithm runs twice (once from A to B and a second time from B to A). It finally returns the maximum of the two which is the trajectory distance. Algorithm 3 takes $O(mn)$ to compute the trajectory distance where n and m are the number of vertices of the input trajectories. This is not an efficient algorithm and there is room for improvement.

There are better ways to use the same technique to compute the trajectory distance. For example, we could store the trajectories in arrays rather than linked lists. Since the order of nodes are preserved, we could perform a binary search on an array to find the position of a given timestamp. This could improve the running time but it is not the optimal solution. Notice that the vertices of the trajectories are in order. We could visit the vertices of the two trajectories in an increasing order of timestamps both at the same time. For example if we have two trajectories: the first one with only two vertices with timestamps 0 and 10 and a second trajectory with 12 vertices with timestamps 0, 1, 2, 3, ..., 11. We could visit the first vertex of the first trajectory and then visit the first 10 vertices of the second trajectory. Next, we visit the second node of the first trajectory and finally the last two vertices of the second trajectory. This technique is very common when it comes to designing algorithms. For example, the famous merge sort algorithm uses a very similar approach when it merges two sorted partitions into a single sorted partition.

Let us examine this approach in more details. First of all, we need to store the trajectories in a different data structure. Linked lists are good structures when data changes frequently. More specifically, they are very efficient when new elements need to be inserted or removed from anywhere in the list. Trajectories are fixed data structures *i.e.*, we generate them once and we never dynamically modify them. In such cases, arrays provide a much better performance compared to linked lists as they provide random access to any element at any index in constant time. When computing the distance between two trajectories, we may query different indexes.

It is important to pick the right data structure for an optimized algorithm. In this case, we choose to store a given trajectory in an ordered array (ordered by timestamps).

We can define an invariant and try to maintain it in our algorithm. Let us assume that we have two trajectories: $P = \{p_1, p_2, ..., p_n\}$ and $Q = \{q_1, q_2, ..., q_m\}$. Let us now define an invariant.

Given two integers i and j such that $0 < i \leq n + 1$ and $0 < j \leq m + 1$, let us assume that the following three statements are true:
1) Let k be a pre-computed value which is the trajectory distance between the two sub-trajectories $\{p_1, p_2, ..., p_{i-1}\}$ and $\{q_1, q_2, ..., q_{j-1}\}$.
2) If $i \leq n$ then $timestamp(p_i) \geq timestamp(q_{j-1})$
3) If $j \leq m$ then $timestamp(q_j) \geq timestamp(p_{i-1})$.

Let us now define some base values for i, j and k such that our invariant holds for any given trajectories. We define k to be the euclidean distance between $position(p_1)$ and $position(q_1)$. Furthermore, let $i = j = 2$. Observe that the three statements above hold for any two trajectories. This is due to the fact that $timestamp(p_1) = timestamp(q_1) = 0$ and it immediately follows that the statements 2 and 3 hold. Furthermore, the trajectory distance between the two sub-trajectories $\{p_1\}$ and $\{q_1\}$ is the euclidean distance between p_1 and q_1. This concludes that the statement 1 also holds. Our goal is to increment i and j one at a time while updating k ensuring our invariant holds.

Let us now address how we can update k while incrementing i or j at each iteration. For now, we assume that $timestamp(p_i) \geq timestamp(q_j)$ and we show how to update i, j and k. Please note that swapping p with q and i with j will work exactly in the same way when $timestamp(p_i) < timestamp(q_j)$. Observe that if $timestamp(p_i) \geq timestamp(q_j)$, then $timestamp(p_{i-1}) \leq timestamp(q_j) \leq timestamp(p_i)$. This means that we need to find the position of trajectory P at timestamp q_j on the edge connection p_{i-1} to p_i. This distance, k', can easily be computed in constant time using the equation (3.10). We can then update k by assigning it to $\max(k, k')$. Furthermore, we increment i by 1. Please observe that after these operations, our invariant will still hold. We continue until either $i = n + 1$ or $j = m + 1$.

Without the loss of generality, let us assume that we reach a point where $i = n + 1$. At this point, the following statements hold due to our invariant:
1) Due to invariant (Statement 3): $\forall_{j \leq j' \leq m}(timestamp(q_{j'}) \geq timestamp(p_n))$
2) Due to invariant (Statement 1): k is the equal to the trajectory distance between P and the sub-trajectory $\{q_1, q_2, ..., q_{j-1}\}$.

Observe that at this point, k accounts for all vertices except for $q_j, q_{j+1}, ..., q_m$ whose timestamps are greater than $timestamp(p_n)$. This means that by iterating over these vertices one at a time, we can update the value of k. For each vertex, we compute its distance with $position(p_n)$. If this distance is greater than of the value of k then we update k with this new distance. At the end of this calculation k is the trajectory distance between P and Q.

Algorithm 4 This algorithm computes the trajectory distance of two trajectories in linear time.

```
Require: $P$: first trajectory given in an ordered array
Require: $Q$: second trajectory given in an ordered array
Ensure: Compute the trajectory distance between $P$ and $Q$
  procedure TRAJECTORYDISTANCE($P$, $Q$)
      if $length(Q) = 1$ then
          SWAP($P$, $Q$)
      end if
      $k \leftarrow euclidean(P[1], Q[1])$
      $i \leftarrow 2$
      $j \leftarrow 2$
      $n \leftarrow length(P)$
      $m \leftarrow length(Q)$
      while $i \leq n \ \& \ j \leq m$ do
          if $timestamp(P[i]) < timestamp(Q[j])$ then
              SWAP($P$, $Q$)
              SWAP($n$, $m$)
              SWAP($i$, $j$)
          end if
          $p \leftarrow (\frac{timestamp(Q[j])-timestamp(P[i-1])}{timestamp(P[i])-timestamp(P[i-1])})\ (position(P[i]) - position(P[i-1])) + position(P[i-1])$
          $k = \max(k, euclidean(Q[j], p))$
          $i = i + 1$
      end while
      while $j \leq m$ do
          $k = \max(k, euclidean(Q[j], P[n]))$
          $j = j + 1$
      end while
      return $k$
  end procedure
```

Algorithm 4 computes the trajectory distance between P and Q in linear time. Observe that each node is only visited once and therefore the running time of the algorithm is $O(n+m)$ where n and m are the number of vertices of the two trajectories. Notice that the algorithm first checks the length of Q. Alternatively, P and Q get swapped if Q only has one vertex.

This is because if Q has only one vertex and the P has more than one vertex, none of the two while loops will execute. As a result, we return the distance between the first vertices of P and Q which is the wrong value. Notice that if both the trajectories only contain one vertex then the result is correct. Moreover, if Q has more than one vertices and P has only one vertex, then the second loop will execute and we compute the correct value. Swapping Q and P in the case where Q has a single vertex ensures that our algorithm always computes the right value. Additionally, notice that in the first while-loop, we first ensure that the timestamp of $P[i]$ is greater that the timestamp of $Q[j]$. If this is not the case, we swap P and Q and their associate indexes i and j and also the variables indicating their length n and m. We can swap these variables in constant time because P and Q are arrays which are pointers in memory and they can be swapped in constant time. Their associate indexes and lengths are only integer variables and they can also be swapped in constant time. This ensures that the algorithm is simple and it follows our invariant.

4.6 Conclusion

In this chapter, we defined our trajectory distance function. Furthermore, we saw that the discrete trajectory distance function is exactly the same as the original trajectory distance function. This observation led us to design a naive algorithm that computes the trajectory distance function in $O(nm)$. We learned that choosing the right data structure to store our trajectories is important. Moreover, using some classical algorithmic techniques borrowed from merge-sort, we were able to design an efficient algorithm that computes the trajectory distance function. This algorithm runs in $O(n + m)$ which is optimal because each node has to be visited at least once for us to compute the discrete trajectory distance. We will use this algorithm for clustering our trajectories in Chapter 6.

Chapter 5

Trajectory Simplification

Chapter 4 covered the trajectory distance function. We also saw how we can compute this distance in linear time. In this chapter, we are going to switch to a completely different topic: The trajectory simplification problem. This chapter explains the reason behind trajectory simplification. In this chapter, we will explain the trajectory simplification problem and we will show why and how it is useful in the context of clustering. Furthermore, we will show two different efficient solutions to this problem. We will compare these two solutions in Chapter 7 in terms of performance and actual running time. The trajectory simplification problem is simple. Given a trajectory T and a threshold $\epsilon \geq 0$, compute a trajectory T' such that $|T'| \ll |T|$ and $\delta_T(T, T') \leq \epsilon$. This problem is very similar to min-# problem introduced in Chapter 2.

5.1 Purpose

The fact is that we already have a linear-time algorithm that can compute the distance between two trajectories. In fact, this linear solution is quite fast in practice. However, trajectories can be very long objects. When we collect trajectory data, the data tends to contain a lot of repetitive information. For example, objects that move slowly tend to accumulate too many vertices for long time intervals and short distances. Even fast-moving objects that maintain a constant speed moving in straight lines normally generate too many vertices on their path that can be eliminated. The problem occurs when the system has to handle a large dataset which contains very long trajectories. Normally, a clustering system needs to capture these trajectories in main memory. One solution is to transfer some of these long trajectories into a physical storage device such as a hard drive when we do not need them and reload them back into memory when necessary. However, there are a few fundamental problems associated with this solution. The first problem is that this approach can weaken the clustering algorithm. Clustering algorithms mostly rely on the objects available to them in main memory. Moreover, longer trajectories mean more trajectories are in the storage device rather than in main memory. As the number of trajectories in the storage device increases, the algorithm operates with a lower ratio of the objects which in turn lowers the quality of the clusters. The second problem is loading information from a storage device into main memory when necessary. Such operations can be expensive for longer trajectories. It takes a longer time to load longer trajectories back into main memory. Observe that for a real-time system, we cannot avoid exporting some of the data into a storage device but if each object is very long then loading them back into memory can be very costly. Most incremental clustering algorithms execute the distance function many times with many objects before finding the appropriate cluster for insertion. If the trajectories contain a large number of vertices, it will take a long time to compute distances between trajectories. This will lead to very slow insertions in the clustering process which is an issue for a real-time system. Consider a new trajectory that has 200 vertices and it is to be inserted into one of the existing clusters in the system. Imagine that the clustering algorithm compares this trajectory with 20 other existing trajectories in the system before finding the right cluster. If each trajectory that the new trajectory is being compared against has 100 nodes then we need to perform 400000 vertex comparisons before the right cluster is found. Now imagine the following scenario: Simplify the new trajectory by a factor of 10 and every single trajectory that already exists in the system was also simplified by the same factor when they were first inserted. In the latter case, we only have to perform 4000 vertex comparisons before finding the right cluster. That means that each insertion can be done 100 times faster. This can be a huge improvement and it really pays off when the number of trajectories increase in the system.

We may be able to save a lot of memory and time if we pre-process a trajectory before inserting it into a cluster. This pre-processing must produce a new trajectory with fewer vertices such that the path and the speed of the new trajectory is not very different than the original trajectory. The goal is to come up with a method such that the number of resulting vertices are much fewer and the trajectory distance between the simplified and unsimplified trajectories is bounded by a threshold. This threshold is introduced in order to ensure that we preserve the quality of our clusters. Let us assume that we have two trajectories A and B and we can compute a simplified version of them A' and B' respectively. The claim is that their respective distance, namely $\delta_T(A', B')$ is almost the same as $\delta_T(A, B)$ where ϵ is a small real number:

$$\delta_T(A, A') \leq \epsilon \tag{5.1}$$

$$\delta_T(B, B') \leq \epsilon \tag{5.2}$$

$$\delta_T(A, B) \leq \delta_T(A, A') + \delta_T(B, B') + \delta_T(A', B') \tag{5.3}$$

$$\delta_T(A, B) \leq \delta_T(A', B') + 2\epsilon \tag{5.4}$$

This is due to the triangulation property. This inequality implies that if we choose a small value for ϵ, the distance between A' and B' is not that different than the distance between A and B. Observe that clustering algorithms attempt to group similar objects in the same cluster. They achieve this by computing the distance between the objects using the distance function. The introduced threshold ensures that distances between objects are preserved and therefore the quality of the final clusters are preserved.

5.2 Remote Trajectory Simplification Method

The remote trajectory simplification aims to attack a very similar problem. It mainly targets Moving Object Databases (MOD). Such databases are designed to store trajectory information of moving objects. The problem is that they normally obtain their data from remote tracking devices such as GPS or mobile devices. These devices are capable of generating a lot of data points in very short time intervals. In these cases, the amount of data can be substential and it can cause two obvious problems. Firstly, these devices transfer their data on wireless communication lines and too many data points causes a waste of bandwidth. The second problem is that MOD must handle large sets of data and it can easily run out of space. There are multiple approaches that attack this problem [32, 21]. Lang and Farrel [22] offer a generic solution to this problem and they call it generic remote trajectory simplification (GRTS). These solutions all have one common ground: They all use a prediction function to predict the next position of the object. Furthermore, they only update the prediction function if and only if the position of the object deviates by a large amount from the predicted value. One important assumption must be made for this approach to work: The time interval between two consecutive data points must be constant.

Although these appoaches are very useful to solve the MOD problems, they make it almost impossible for us to use our distance function. These approachs make it easy to store trajectory data points in database using multiple prediction functions. Unfortunately, they make it really hard to load these data points in memory with less data points. Thus, they will not improve the running time of trajectory distance function.

5.3 Approach

We have already seen in Chapter 2 that there are numerous solutions for the min-# problem. The trajectory simplification problem is a very similar problem to min-# problem. With all the similarities, we must be careful because there are also many differences. The min-# problem is defined over polygonal chains rather than trajectory objects. Polygonal chains have no information about time or speed of an object moving through space. The trajectory distance function is dependant on both the coordinates and the timestamps of the spatio-temporal nodes.

Let us start with a greedy approach to solve this problem. Consider a trajectory obtained from tracking a person using a surveillance system. Furthermore, let us assume that the person is walking on an approximately straight line at a constant speed. Moreover, the tracking system is capturing the position of the person in space 10 times per second. If the person takes two steps a second, then the tracking system constructs a trajectory with 5 spatio-temporal nodes for every step. Thus, if the system tracks the person for only 10 seconds then a trajectory with 100 spatio-temporal nodes gets generated. Notice that the person is moving at a constant speed on an approximately straight line. This implies that all the nodes except for the first and the last node can be eliminated. Observe that this elimination of the nodes in the middle will result in a trajectory with only two nodes. This new trajectory is defined on approximately the same path and at the same speed as the original trajectory. Figure 5.1 illustrates an example of such simplification.

There is a clear problem with this approach. This approach would only work if our trajectories only move on relatively straight lines at constant speeds. This is actually not true in practice. Moving objects change direction and their velocity may fluctuate as they move through space. Therefore, this is clearly a very unrealistic assumption. In order to resolve this issue, we could perform the same method until the trajectory changes direction or speed.

This chapter will cover two methods to solve the simplification problem. The second method is a slight modification of the solution given for simplifying polygonal chains under Fréchet distance [2]. This approach was addressed in Chapter 2. The algorithm starts from the initial node of a trajectory and iterates through its nodes one by one. At each step, it computes the smallest enclosing disk that contains all these nodes. If the radius of the enclosing disk is less than the threshold ϵ then the algorithm moves onto the next point. On the other hand, if the radius exceeds the threshold, it takes a step back and removes the last node. In most cases two nodes are inserted to the simplified version of the trajectory. Both of these nodes are located at the centre of the enclosing disk with two different timestamps in order to synchronize the speed with the original trajectory. We will cover this algorithm in details in this chapter.

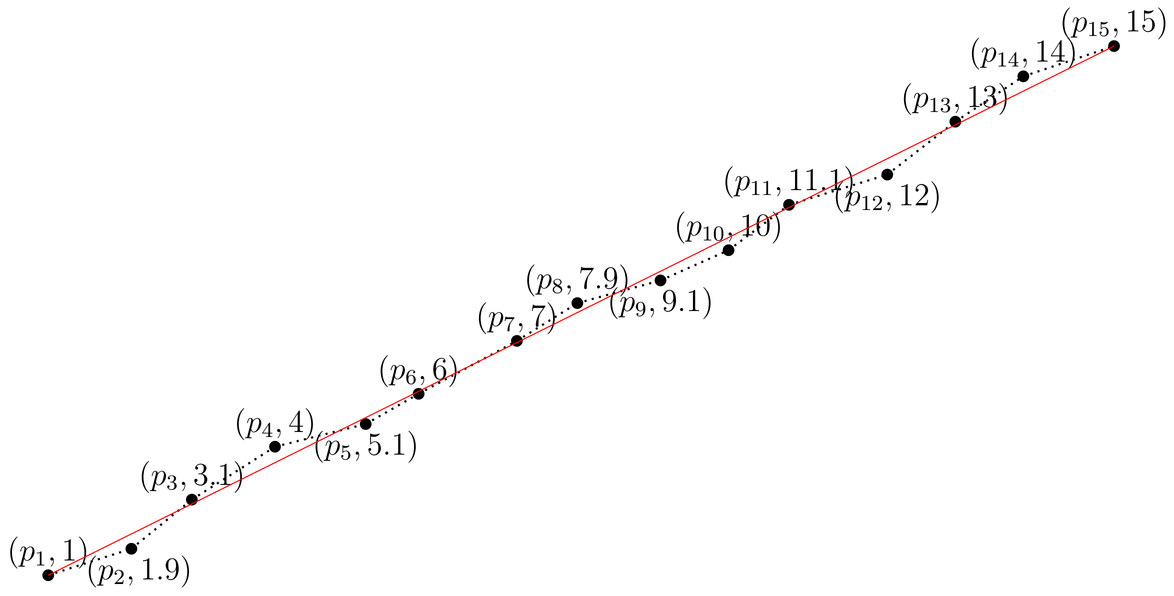

Figure 5.1: This figure illustrates how trajectory information can become long and how they can be simplified with minimal loss of information.

5.4 Doubling Search Method

This section covers a method that can be exploited to design algorithms to perform efficient searches. We will use this method in both of our solutions in this chapter. Let us first give a general introduction of the search problems that can be solved using the doubling search method. Assume that an ordered list $L = \{o_1, o_2, ..., o_n\}$ containing n objects is given as input. The order of the objects in the list is important and changing the order of the objects will affect the representation given by the list.

Observe that trajectories and polygonal chains share this property. The nodes in a trajectory must be in the order in which points appear through space. Changing the order of nodes will affect the representation of a trajectory. In fact, changing the order of the nodes has broader consequences when it comes to trajectories. Trajectories are required by definition to have nodes whose timestamps progresses sequentially. This means that switching two nodes will cause a trajectory to violate this rule which in turn will make it invalid by definition.

In addition to the ordered list, a rule R is also given to us as an input. This rule is either satisfied or not satisfied for the first k elements of an ordered list L. The rule R must have the following three properties:

1. R is always satisfied for the first element of L

2. If R is satisfied for the first i elements of L then it must be satisfied for the first j elements where $0 < j \leq i$

3. If R is not satisfied for the first i elements of L then it also is not satisfied for the first j elements such that $j \geq i$

R is a mathematical tool but in order to address our search problem algorithmically, we need to replace R with a function. Let $f_R(L,k)$ be a function that is mapped to "true" if R is satisfied by the first k elements of the list L and otherwise, it is mapped to "false":

$$f_R(L,k) = \begin{cases} true & \text{if } R \text{ is satisfied for the first } k \text{ elements of } L \\ false & \text{if } R \text{ is not satisfied for the first } k \text{ elements of } L \end{cases}$$

We also assume that the evaluation of f_R has a running time complexity of $\theta(k)$. Our task is to compute the largest index i such that the rule R is satisfied for the first i elements of the list. Once the largest index i is found, consider a new list $L' = \{o_{i+1}, o_{i+2}, ..., o_n\}$. We would like to continue our search on this new list and do this until the list is empty. Let us give an example to clarify this. Consider an ordered list of integer numbers $L = \{1,2,3,2,4,1,5,2\}$ and the following rule R: given an integer k and an ordered list L, the rule R is only satisfied if the sum of the first k elements of R is at most 7. The sum of the first three elements of the list L is 6 but the sum of the first four elements of L is 8 therefore the first largest index is 3. At this point, we are left with a shorter list: $\{2,4,1,5,2\}$. Although at each step our list gets smaller but we still use the indexes of the original list L. In the new shorter list, the first three elements of this list satisfy the rule R but not the first four elements. Therefore the second largest index is 6. We are now left with the following list: $\{5,2\}$. In the last step, both of the elements satisfy the rule R and therefore the last index that we find is 8. These indexes are placed into an ordered set called "the optimal index set": $I = \{3,6,8\}$.

Let us now examine an algorithm that can solve this problem. Our goal is to compute the optimal index set for an input ordered list L of size n and an input function $f_R(L,k)$. The only assumption that we will make is that the input list to the algorithm is in an array data structure. Let us first start by a greedy and naive algorithm. The simplest way to solve this problem is to iterate over the indexes of the list from the first index to the last index. This can be done using a simple for-loop with an incremental index i. The algorithm does not need to check for the first index of the list because according to the first property of R, $f_R(L,1)$ must evaluate to true. Therefore, the algorithm start from the second index of the list. Once f_R evaluates to false then $i-1$ is the largest index in L for which R is satisfied. If the algorithm exits the for-loop without returning an index then the largest index for which R is satisfied is n. Thus, the algorithm returns n if it exits the for-loop. This will only compute the largest index of L

that satisfies R. Recall that our goal is to continue until the list is empty. We can do that using a second algorithm (Refer to algorithm 6).

Algorithm 5 Returns the largest index for which rule R is satisfied.

```
Require: $L$: an array of elements
Require: $n$: number of elements in $L$
Require: $f_R$: a linear running time function that replaces rule $R$ as described.
  procedure GreedyLargestIndexSearch($L$, $n$, $f_R$)
    for $i \in \{2, ..., n-1\}$ do
      if $f_R(L, i)$ evaluates to false then
        return $i - 1$
      end if
    end for
    return $n$
  end procedure
```

Algorithm 6 always terminates. This is because algorithm 5 will always return an index of at least 1 and that is due to the first property of R. Observe that algorithm 6 is not calling the function in algorithm 5. This is because we are going to modify algorithm 5 but for now we assume that it is going to call the function in algorithm 5. Let us now explain how algorithm 6 works.

The algorithm computes the optimal index set $I = \{i_1, i_2, ..., i_m\}$ which contains indexes of L with the following properties:

1. i_1 is the largest index in L that satisfies the rule R.
2. For any $j > 1$, the index i_j is the largest index in $L' = \{o_{i_{j-1}+1}, o_{i_{j-1}+2}, ..., o_n\}$ that satisfies the rule R.

Algorithm 6 calls the function in algorithm 5 at each step in the while-loop to compute the next index. The algorithm updates the list L and its size, n, at each iteration. The problem with this algorithm is its quadratic running time complexity assuming f_R has a linear time solution. This is very inefficient for our purposes. As we will see, our simplification algorithm depends on an algorithm that computes the optimal index set. We cannot use algorithm 6 as a foundation for our simplification solution due to its inefficiency.

In order to solve this problem we use the doubling search method [18, 2, 4]. Observe that algorithm 6 can run faster if algorithm 5 can be improved to run faster. This is exactly what

Algorithm 6 Runs algorithm 5 Computes the optimal index set I

```
Require: $L$: an array of elements
Require: $n$: number of elements in $L$
Require: $f_R$: a linear running time function that replaces rule $R$ as described.
  procedure ComputeOptimalIndexSet($L$, $n$, $f_R$)
      $I \leftarrow \{\}$
      $j \leftarrow 0$
      while $n > 0$ do
          $i \leftarrow$ LargestIndexSearch($L$, $n$, $f_R$)
          $j = i + j$
          $I = I \cup \{j\}$
          $L = \{o_{j+1}, o_{j+2}, ..., o_n\}$
          $n = n - i$
      end while
      return $I$
  end procedure
```

we are going to do here. We are going to modify algorithm 5 by performing two searches instead of a single linear search. Observe that the running time complexity of algorithm 5 is $\theta(i^2)$ where i is the computed index that is returned by this algorithm. This is simply because of the linear running time complexity of f_R.

The doubling search method has two phases. First, the algorithm finds a range in the list that contains i. Once this range is computed, a binary search is performed to find the exact index i. In the first search phase, the algorithm looks for the largest value for t such that:

1. Rule R is satisfied by the first 2^{t-1} elements of the list

2. Rule R is not satisfied by the first 2^t elements of the list

Once the algorithm finds the value of t, it will perform a binary search to find i within the range of indexes from 2^{t-1} to 2^t.

Algorithm 7 demonstrates how the doubling search method works. Let us first examine the algorithm and then we will derive the running time complexity of the algorithm. Observe that the algorithm initializes t to 1. In other words, the first index that is to be checked is the second index. This is again because we know R is satisfied for the first index of L. The algorithm performs its first search first in the while-loop. Furthermore, at each iteration of the

Algorithm 7 Finds the index i in two search phases.

```
Require: $L$: an array of elements
Require: $n$: number of elements in $L$
Require: $f_R$: a linear running time function that replaces rule $R$ as described.
  procedure DOUBLINGSEARCH($L$, $n$, $f_R$)
      $t \leftarrow 1$
      while $2^t < n \ \& \ f_R(L, 2^t)$ evaluates to true do
          $t = t + 1$
      end while
      $m \leftarrow 2^{t-1}$
      $k \leftarrow \min(2^t, n)$
      Perform a binary search in $L[m, m+1, ..., k]$ to find the index $i$
      return $i$
  end procedure
```

while-loop, the value of t is incremented until either 2^t becomes larger than the size of L or $f_R(L, 2^t)$ evaluates to false. Given any of those conditions, the algorithm exits the loop and performs a binary search within the computed range. It finally returns the index found from the binary search.

We shall now compute the running time complexity of this algorithm. From now on, we assume that algorithm 6 calls the function in algorithm 7 to compute the largest index. Let m be the size of the optimal index set computed by algorithm 6 when the algorithm terminates. Furthermore, let k_i be the i^{th} index returned by the algorithm. Let us focus on k_1 for now and then we will apply our method for the rest of the indexes. The first search is done in $O(k_1 \log k_1)$. This is due to the fact that k_1 must be between 2^{t-1} and 2^t. This implies that $k_1 < 2^t$. In other words, the loop in the doubling search function iterates $\lceil \log k_1 \rceil$ times. Furthermore, the binary search should also terminate in $\lceil \log k_1 \rceil$ iterations. In each iteration, the binary search computes f_R which runs in at most $\theta(2k_1)$ time. Therefore the first breaking point k_1 is found in $O(k_1 \log k_1)$. Moreover, since we remove the first k_1 elements, the same applies to k_2 and so on. The running time of the algorithm can be expressed as following:

$$T(n) = k_1 \log k_1 + k_2 \log k_2 + ... + k_m \log k_m \tag{5.5}$$

Furthermore, we know that the sum of all the indexes will be equal to n:

$$\sum_{i=1}^{m} k_i = n \tag{5.6}$$

This concludes that the running time of the function above is $O(n \log n)$.

5.5 Direct Link Simplification Method

In Section 5.4, we introduced the notion of an abstract rule that must have three properties. In this section, we will modify the doubling search method slightly. Moreover, we define a concrete rule for the doubling search method to simplify a trajectory. We will see that this rule does not obey the last two properties introduced in Section 5.4. Let us first examine the consequences of removing the last two properties of a rule. Observe that we need the first property to ensure the termination of algorithms 6 and 7. However, properties two and three are only there to ensure that the output of the algorithm is always consistent and easily predictable. Let A be an array of abstract elements $\{a_1, a_2, ..., a_n\}$. Moreover, consider a second array of elements that corresponds to values observed by running the rule function $f_R(A, i)$. Let us call this the ruling array and denote it with $\lambda = \{r_1, r_2, ..., r_n\}$. Clearly, the domain of the ruling array is a binary set: $\{true, false\}$. Furthermore, for any $0 < i \leq n$, we know the following: $r_i = f_R(A, i)$. The doubling search method assumes one of the following outcomes for λ:

1. $\lambda = \{true, true, true, ..., true\}$
2. $\lambda = \{true, true, ..., true, false, false, ..., false\}$

The first outcome is the case where function f_R evaluates to $true$ for any given index of the array A. In this case, the largest index that satisfies rule R is the size of the array which we denoted by n. Therefore the optimal index set is $I = \{n\}$. The first element of λ is always true due to the first property of R. The second outcome above has a breaking point. In other words, the second outcome breaks λ into two consecutive partitions. The first partition is all the ”true” values and the second partition is all the ”false” values.

Let us now assume that we have a rule that does not obey properties two and three but it always obeys the first property. As a result of this assumption, the array λ can have outcomes like the following:

$$\{true, true, false, false, true, false, true, false, false, ...\}$$

Recall that our new rule still obeys property 1: the first element of λ is always $true$. We cannot have any further assumptions on the outcome of λ. In other words, λ can be partitioned into more than two consecutive $true$ or $false$ values. A breaking point occurs when a $true$ value is followed by a $false$ value in λ. Therefore, we can also have multiple breaking points.

Removing the last two properties of R can make λ have no breaking points or 1 or more breaking points. This does not mean that the doubling search algorithm will no longer work. The only consequence is that it is not easy to predict which breaking point the doubling search function returns at each iteration. The algorithm will still execute its task and it is guaranteed that it will terminate due to the first property of the rule R.

Let us now define our concrete rule which we will call the direct link rule. We define the function $f_{dl}(T, i)$ that corresponds to rule R_{dl}:

Definition 5.1 (Direct link rule function). G*iven a trajectory* $T = \{c_1, c_2, ..., c_n\}$*, an index* i *and a threshold value* $\epsilon \geq 0$*, the direct link rule function is defined as following:*

$$f_{dl}(T, i) = \begin{cases} true & \textit{if } \delta_T(T_i, T') \leq \epsilon \\ false & \textit{if } \delta_T(T_i, T') > \epsilon \end{cases}$$

where $T' = \{c_1, c_i\}$ *and* $T_i = \{c_1, c_2, ..., c_i\}$

Recall that in Chapter 4, we presented the trajectory distance algorithm that has a linear running time complexity. Hence, the direct link ruling function $f_{dl}(T, i)$ has a running time complexity of $\theta(i)$. Same distance function can be used to compute the direct link rule function. In this case, trajectory T_i has a length of i and T' has a length of two. Using this approach, the total running time will be $\theta(i + 2) = \theta(i)$. We conclude that the running time complexity of the function f_{dl} is $\theta(i)$. This is required to ensure that the doubling search algorithm preserves its running time complexity of $O(n \log n)$.

Furthermore, the direct link rule obeys the first property: it always evaluates to $true$ for any input trajectory and $i = 1$. Consider an input trajectory $T = \{c_1, c_2, ..., c_n\}$ and an input index i which is equal to 1. As a result of this input, we can construct T_i and T': $T_i = T_1 = \{c_1\}$ and $T' = \{c_1, c_1\}$. In this case, T' can be simplified. This is because the first and the second node of T' are exactly the same and therefore they have the same coordinates and timestamps. In other words, we have the following $T' = \{c_1, c_1\} = \{c_1\}$. This implies that $T' = T_1$:

$$\delta_T(T_1, T') = 0 \tag{5.7}$$

The threshold ϵ is a non-negative value. This brings us to the conclusion that $f_{dl}(T, 1)$ always evaluates to $true$ for any given input trajectory T. This confirms that the direct link rule obeys the first property that is required to ensure the termination of the doubling search algorithm.

Let us now address one of the properties of the trajectory distance function. The trajectory distance function is the maximum distance between any two input trajectories at the timestamps defined on the nodes of the trajectories. This observation leads us to the realization of a property of the distance function that can be exploited for the simplification problem.

Recall that the distance function was defined as $\delta_T(T_1, T_2) = \max_{t \geq 0}(d_{T_1T_2}(t))$ where $d_{T_1T_2}(t)$ is the euclidean distance between T_1 and T_2 at time t. This implies that for some timestamp α, $\delta_T(T_1, T_2) = \max_{t \geq 0}(d_{T_1T_2}(t)) = \max(\max_{0 \leq t \leq \alpha}(d_{T_1T_2}(t)), \max_{t \geq \alpha}(d_{T_1T_2}(t)))$. We can think of α as a time breaking point that partitions each trajectory into two trajectories. The trajectory distance function can be applied to these partitions in order to compute the overall trajectory distance function. The trajectory distance is defined as the maximum of the trajectory distances of each partition. This observation can be extended into more than two partitions: Let $\alpha_1 < \alpha_2 < ... < \alpha_m$ be m distinct points of time. The trajectory distance between any two trajectories T_1 and T_2 can be re-defined as:

$$\max(\max_{0 \leq t \leq \alpha_1}(d_{T_1T_2}(t)), \max_{\alpha_1 \leq t \leq \alpha_2}(d_{T_1T_2}(t)), ..., \max_{\alpha_{m-1} \leq t \leq \alpha_m}(d_{T_1T_2}(t)), \max_{t \geq \alpha_m}(d_{T_1T_2}(t))) \tag{5.8}$$

We use this property of the distance function to solve our simplification problem. These time points α_i partition an input trajectory into multiple trajectories. Figure 5.2 clarifies the concept.

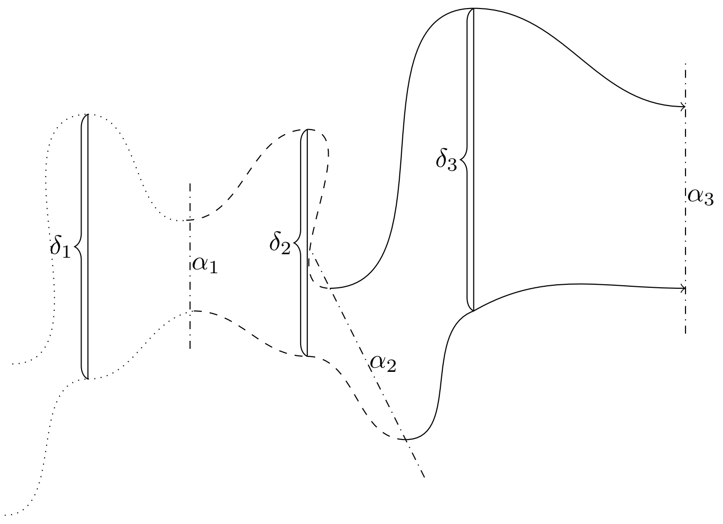

Figure 5.2: Illustrates the trajectory distance function computed in multiple partitions. The final trajectory distance function can be computed by taking the maximum distance of all the partitions. In the case above, two trajectories are partitioned into three partitions using three time points (α_1, α_2, α_3). Each partition is compared against its corresponding partition in the other trajectory resulting in three distances (δ_1, δ_2, δ_3). The trajectory distance between the two actual trajectories is the maximum of these three distances which in this case is δ_3.

Let $T = \{c_1, c_2, ..., c_n\}$ be a trajectory that is to be simplified. We take the following approach to simplify the trajectory. Partition T at different time points $\alpha_1, \alpha_2, ..., \alpha_m$ into partitioned trajectories $T_1, T_2, ..., T_m$. Observe that α_m is the timestamp of c_n. Next, replace each new partitioned trajectory T_i with another sub-trajectory $T_i{}'$ such that $\delta_T(T_i, T_i{}') \leq \epsilon$. In order to compute the partitions, we are going to apply the direct link rule with some arbitrary ϵ to the doubling search algorithm. This will result in an optimal index set $I = \{i_1, i_2, ..., i_m\}$. This optimal index set can be used to compute replacements for each partitioned trajectory.

We are first going to examine the optimal index set that we obtain from this approach with an example. Let $T = \{c_1, c_2, ..., c_{10}\}$ be a trajectory that we are going to simplify. Assume that we obtain $I = \{3, 6, 10\}$ as the optimal index set after we execute the doubling search method using the direct link rule function. This optimal index set splits T into three partitions:

- $T_1 = \{c_1, c_2, c_3\}$
- $T_2 = \{c_4, c_5, c_6\}$
- $T_3 = \{c_7, c_8, c_9, c_{10}\}$

Given these partitions, we can compute three other trajectories where each corresponds to one partition. This computation is very simple and we can obtain these trajectories by removing all the nodes from each partition except for the first and the last node:

- $T_1{}' = \{c_1, c_3\}$
- $T_2{}' = \{c_4, c_6\}$
- $T_3{}' = \{c_7, c_{10}\}$

Due to the direct link rule, the trajectory distance between each partition and its corresponding two-member trajectory is at most ϵ. Therefore, we can merge the corresponding trajectories to obtain a simplified trajectory for T: $T' = \{c_1, c_3, c_4, c_6, c_7, c_{10}\}$.

5.6 Enclosing Disk Simplification Method

The second solution for the trajectory simplification problem is based on the smallest enclosing disk problem. The 2D smallest enclosing disk problem can be defined as follows: Given a set containing n points in R^2 plane, compute the smallest enclosing disk that covers all the points in the set. This is a classic problem with a very long history. The problem was first proposed back in 1857 by Sylvester [30].

5.6.1 Smallest Enclosing Disk Problem

We will first make three general claims about smallest enclosing disks and prove them. Please note these are known proofs [33]. Furthermore, we will use these three claims to design a greedy algorithm to solve the smallest enclosing disk problem. We will show that the greedy algorithm has a polynomial running time complexity.

The first claim is that the smallest enclosing disk for a set of points P in R^2 is always a unique disk: We use contradiction to prove this claim. Let $P = \{p_1, p_2, ..., p_n\}$ be a set of distinct points in R^2. Consider two distinct disks C_1 and C_2 covering the points in the set P. Let us assume that C_1 and C_2 are both smallest enclosing disks for the set P and therefore they have the same radius r. Furthermore, let c_1 and c_2 be the centres of the disks C_1 and C_2 respectively. Observe that the points in P must be in the intersection of C_1 and C_2 for them to be covered by both the disks. However, $C_1 \cap C_2$ can be covered by a smaller disk C whose centre is positioned at $\frac{c_1+c_2}{2}$. Furthermore, the radius of C is equal to $\sqrt{r^2 - x^2}$ where x is half the distance between c_1 and c_2. Observe that radius of C is smaller that r (Refer to Figure 5.3).

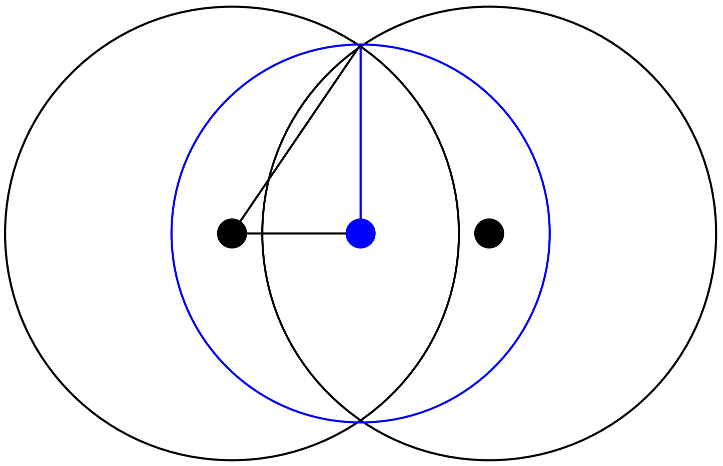

Figure 5.3: This figure shows if two distinct disks of the same size enclose a set of points, then the points must exist in their intersection. Furthermore, the intersection of the two disks is contained in a smaller disk. The radius of the smaller disk can simply be calculated using the Pythagorean theory.

The second claim is for a set P with at least two distinct points: Let P be a set with at least 2 distinct points in R^2. We make the following claim: The circumference of the smallest enclosing disk covering the points in P must cover at least two points in P. Imagine a disk C covering all the points in P whose circumference does not cover any of the points in P. We can continuously reduce the radius of C until a point in P is covered by the circumference of the smallest enclosing disk of P. This observation confirms that at least one point in P must be covered by the circumference of C. Now, imagine a disk C whose circumference only covers one point p in P. We can continuously move the centre of the disk C toward the point p on the circumference until a second point appears on the circumference of the disk. This second observation confirms our second claim.

We finally make the last claim here: Let $P = \{p_1, p_2, ..., p_n\}$ be a set of points in R^2 and let C be the smallest enclosing disk that covers the points in P. Furthermore, let P_C be a subset of P that contains all the distinct points that lie on the circumference of C. We make our final claim: If the set P_C contains at least 3 distinct points, the smallest enclosing disk, C, can be computed as a function of any three distinct points in P_C. Given any three distinct co-circular points in R^2, there exists a unique disk whose circumference covers these three points. Let A and B be two distinct points in R^2 plane. Observe, that the set of points that are equidistant from A and B define a unique straight line l_{AB} in R^2 plane that goes through the midpoint of A and B. Imagine a third point C that is co-circular with A and B *i.e.*, A, B and C are not co-linear. Observe that l_{AB} and l_{BC} are not parallel and therefore intersect each other exactly at one point c in R^2 plane. This point, c, is the centre of the disk whose circumference covers the points A, B and C. Observe that there exists exactly one such point and therefore there is exactly one disk whose circumference covers these points. The radius of the disk is the euclidean distance between c and any of the points A, B or C.

We may now use these three claims to design a greedy algorithm to solve the smallest enclosing disk problem: Let $P = \{p_1, p_2, ..., p_n\}$ be a set of n distinct points in R^2. Observe that if P contains only one point, p, then the smallest enclosing disk has a radius of 0 and it is centred at point p. We now focus on the problem where P contains more than one point. Let C_{pair} be the set of all the smallest enclosing disks of any distinct pair of points in P. These disks are very simple to compute: Given two distinct points p and q in R^2, the centre of their smallest enclosing disk, c, is the midpoint of p and q. The radius of the disk is half the distance between p and q. Observe that the computation of each disk has a constant running time complexity. There is a total of $\binom{n}{2}$ of such disks and therefore the computation of C_{pair} has a $O(n^2)$ running time complexity. Furthermore, let $C_{triplet}$ be the set of all the disks whose circumferences cover any three distinct co-circular points in P. Given any three points in R^2, it takes $O(1)$ to check if they are not co-linear (therefore they are co-circular). If the points are co-circular, we can use the method described to prove the third claim to compute the centre of a disk whose circumference covers the three points. This method also has a constant running time complexity. Observe that the maximum number of such disks is $\binom{n}{3}$ (some triplets may not be co-circular). Thus, the computation of $C_{triplet}$ has a $O(n^3)$ running time complexity.

We know that there is exactly one smallest enclosing disk covering all the points in P from the first claim. The second and the third claim suggest that the smallest enclosing disk exists in $C_{pair} \cup C_{triplet}$. We can simply iterate over these disks and keep track of the smallest disk that covers all the points in P. Observe that there is a total of $O(n^3)$ disks to inspect. Each inspection has a linear time complexity and therefore we spend $O(n^4)$ to compute the smallest enclosing disk. This is obviously not an efficient solution. However, it implies that the problem can be solved in polynomial time and therefore it is not NP-hard.

The smallest enclosing disk problem has a very long history after it was first introduced in 1857. This problem received a lot of attention among researchers but surprisingly no efficient solution was proposed until 1975. In 1975, Shamos and Hoey [28] proposed the first efficient and practical solution to this problem which runs in $\theta(n \log n)$. This solution depends on constructing the Voronoi diagram of n points in the plain. In fact, it is a special type of Voronoi diagram called the FPVD (farthest-point Voronoi diagram). FPVDs are powerful structures that help in solving numerous geometric problems. The construction of FPVD has a lower bound running time complexity of $\Omega(n \log n)$. He concluded that the smallest enclosing disk problem has a lower bound running time complexity of $\Omega(n \log n)$. However, this was the wrong conclusion as a linear solution was introduced seven years later [24].

We are going to give an overview of Shamos's and Hoey's approach here because it is a classical solution to the smallest enclosing disk problem. Let $P = \{p_1, p_2, ..., p_n\}$ be a set of points in R^2. The FPVD diagram partitions the plain into regions V_i for each point p_i. Each region V_i can be defined as a set of point such that:

$$q \in V_i \iff \forall_{j \in J}(|qp_i| > |qp_j|) \text{ where } J = \{1, 2, ..., n\} - \{i\} \tag{5.9}$$

We are not going into the details of this algorithm. The basic idea is that to compute the FPVD of a set of points P. Next, use FPVD to compute the centre of the smallest enclosing disk of P.

The running time complexity of computing FPVD diagrams for a set with n points is $\theta(n \log n)$. An FPVD can then be used to find a pair of points p_i, p_j in P with the maximum distance between them. The furthest pair of points, $p_i, p_j \in P$ can be computed from FPVD in linear time. Observe that the pair of points (p_i, p_j) is not necessarily a unique pair. This pair of points define a disk C whose diameter equals to $|p_i p_j|$ and its centre is the mid-point of p_i and p_j. Thus, C can be computed in constant time given p_i and p_j. It can be shown that the smallest enclosing disk of P is either C or it is a disk whose centre is at a point where three Voronoi regions meet. It can also be shown that there exists at most $n - 2$ such points in the diagram. The centre of the smallest enclosing disk, c, can be computed in linear time after computing the FPVD. In fact, the challenge is to find the centre of the smallest enclosing disk. Once this point is computed, the smallest enclosing disk can be computed in linear time by iterating over the points in P. The algorithm proposed by Shamos and Hoey [28] computes FPVD in $\theta(n \log n)$, Finding the centre of the enclosing disk and computing the enclosing disk will take linear time once FPVD is computed. Therefore, his algorithm has a running time complexity of $\theta(n \log n)$.

The actual procedure is very complex and we are not going to use it as part of our solution here. Shamos and Hoey [28] concluded that this was the best we can do in finding the smallest enclosing disk. That is, he concluded that this problem has a lower bound of $\Omega(n \log n)$. It is important to note that this is not a lower bound and his approach caused him to draw the wrong conclusion.

Exactly seven years later, Meggido [24] translated this problem into a linear programming problem which can be solved in linear time. Meggido's approach consists of two parts. In the first part, a constrained version of the smallest enclosing disk is introduced. The constrained version of the problem makes an attempt to force the centre of the smallest enclosing disk on a line L in the plain. After solving the constrained problem, the algorithm can determine if the centre of the disk lies on L. Should the centre of the disk not lie on L, the algorithm can determine which side of L it must continue its search. This becomes a basis for the second part of this solution which is the unconstrained part of the problem. Nimrod shows that both parts can be solved in linear time and therefore there is a linear algorithm for the smallest enclosing disk problem. This linear solution is optimal because each point in P has to be visited at least once before the enclosing disk can be computed. However, this solution is extremely complex and it is not very intuitive to implement. We only mention Nimrod's solution to point out that there exists an absolute linear solution to this problem. In this chapter we use Welzl's approach [33] to solve the smallest enclosing disk problem. Welzl's solution is randomized and it has an expected linear running time. We use this approach for two main reasons. First of all, to simplify a large number of trajectories, we will run this algorithm so many times for so many inputs. Since, the algorithm is executed on so many input sets, the linear expected running time will converge toward a real linear time algorithm overall. The second reason is its simplicity. Welzl's approach [33] is extremely simple to implement compared to Nimrod's approach [24].

5.6.2 Randomized Solution

In 1991, Welzl introduced an algorithm [33] that computes the smallest enclosing disk covering a set of points P in linear expected time. The algorithm uses the properties of the smallest enclosing disk introduced in Section 5.6.1. This section covers Welzl's algorithm. Given a set P of n distinct points in the plane, let $md(P)$ be the closed disk of smallest radius covering all the points in P. Notice that $md(P) = \emptyset$ when $P = \emptyset$ and also $md(P) = \{p\}$ when $P = \{p\}$. If $md(P)$ has exactly two distinct points from P on its boundary, the distance between those points is the length of the diameter of the circle. If there are more than two points of set P on the boundary of $md(P)$, any combination of three points from those points on the boundary can define $md(P)$. The centre of $md(P)$ will be the circumcenter of any triangle formed by any three points on the boundary.

These are all the points we covered in Section 5.6.1. Welzl's achievement comes from one important observation: There is a subset $S \subseteq P$ of points on the boundary of $md(P)$ such that $|S| \leq 3$ can form the smallest enclosing disk: $md(P) = md(S)$. This implies that if $p \notin S$, then $md(P - \{p\}) = md(P)$, or if $md(P - \{p\}) \neq md(P)$ then $p \in S$.

Let us first start with an incremental algorithm. The algorithm starts with an empty set as the solution. Moreover, the algorithm processes a new point from the set maintaining the smallest enclosing disk at every step. Let $P = \{p_1, p_2, ..., p_n\}$ be the input set of points to this algorithm. The algorithm has the following invariant: at the i^{th} step of the algorithm, the smallest enclosing disk D_{i-1} of points $\{p_1, p_2, ..., p_{i-1}\}$ has already been computed. Furthermore, the algorithm makes an attempt to process p_i. In other words, the algorithm computes the smallest enclosing disk D_i of points $\{p_1, p_2, ..., p_i\}$ at its i^{th} step. The algorithm first checks if $p_i \in D_{i-1}$, then $D_i = D_{i-1}$. That is, we have already computed the smallest enclosing disk for the set of points $\{p_1, p_2, ..., p_i\}$ and we are done with the i^{th} step. This is due to the property we introduced above. However, if $p_i \notin D_{i-1}$ then this means that p_i is on the boundary of the enclosing disk of D_i. Let us assume that we have a helper function called $b\_minidisk(A, p)$ where A is a set of points and p is a point. $b\_minidisk(A, b)$ computes the smallest enclosing disk covering all the points in $A \cup \{p\}$ such that p is on the boundary of this disk.

Algorithm 8 is recursive. It may be simpler to present this algorithm using a for-loop that would iterate through the points in P. However, each step relies on the output of the previous step. Both the recursive and the iterative approaches would be the same but it is easier to analyze the time complexity of this algorithm when it is recursive. Moreover, the final algorithm in this section is a modification of Algorithm 8 which is recursive. It is worth noting that both the recursive and the iterative algorithms would have the same running time.

Algorithm 8 Computes the smallest enclosing disk that covers all the points in P

```
Require: $P$: set of points to be covered by the resulting disk
  procedure MINIDISK($P$)
      if $P = \emptyset$ then
          $D := \emptyset$
      else
          $p \in P$ ; where $p$ is chosen randomly
          $D :=$MINIDISK($P - \{p\}$)
          if $p \notin D$ then
              $D :=$B_MINIDISK($P - \{p\}, p$)
          end if
      end if
      return $D$
  end procedure
```

The running time of this algorithm depends on the running time of $b\_minidisk(A, p)$. Let us consider this function as a black box and assume that it needs $c\,|A|$ steps to perform its computation. Note the point p is chosen completely randomly from the set P at each step. Let $t(n)$ be the expected running time of our algorithm:

$$t(n) \leq 1 + t(n-1) + Prob(p \notin md(P - \{p\}))c(n-1) \tag{5.10}$$

The inequality above is quite obvious. It consists of three parts:

- 1: accounts for the constant work required
- $t(n-1)$: accounts for the recursive call to $minidisk$
- $Prob(p \notin md(P - \{p\}))c(n-1)$: accounts for call to $b\_minidisk$

Notice that at each step, there are at most 3 points for which $p \notin md(P - \{p\})$. Thus, $Prob(p \notin md(P - \{p\})) \leq \frac{3}{n}$. Therefore the last term can be replaced by $\frac{3}{n}c(n-1)$:

$$t(n) \leq 1 + t(n-1) + \frac{3}{n}(c(n-1)) \tag{5.11}$$

We conclude that $t(n) \leq (1+3c)n$. In other words, the expected running time of this algorithm is $O(n)$. This expected running time depends on a subroutine $b\_minidisk(A, p)$ that requires $c\,|A|$ steps to perform its task. We have not given a description of this algorithm and it remains to be a black box. This subroutine also follows along the same lines and it requires another subroutine that computes the smallest enclosing disk of a set of points with two points on the boundary of the enclosing disk. Furthermore, this new subroutine will require another subroutine that computes the smallest enclosing disk of a set of points with three points on the boundary of the enclosing disk. It turns out that we can combine $minidisk(P)$ and all of these subroutines into a single procedure which we will cover later. Our next algorithm depends on some mathematical theorems that we present here.

For two finite sets of points P and R, let $b\_md(P, R)$ denote the smallest enclosing disk that contains all the points in P and all the points in R on its boundary. Note that $b\_md(P, \emptyset) = md(P)$. Moreover, note that if R is a non-empty set then $b\_md(P, R)$ may be undefined. Given the following two finite sets where P is a non-empty set and $p \in P$, we make three claims:

1. If $b\_md(P, R)$ is defined then it is a unique disk.
2. If $p \notin b\_md(P - \{p\}, R)$ then p lies on the boundary of $b\_md(P, R)$, provided it exists: $p \notin b\_md(P - \{p\}, R) \rightarrow b\_md(P, R) = b\_md(P - \{p\}, R \cup \{p\})$
3. If $b\_md(P, R)$ exists then there is a set S of at most $\max\{0, 3 - |R|\}$ points in P such that $b\_md(P, R) = b\_md(S, R)$

The proof of the theorems above can be found in [33]. These three theorems are the basis for combining all the subroutines into a single procedure. Algorithm 9 has two base cases. The algorithm returns the disk with points R on the boundary if the set P is empty or the set R has three points on it. Notice that if the procedure is initiated with a call where $R = \emptyset$ then for all the recursive calls inside the function, the cardinality of R is at most 3. Furthermore, notice that $b\_md(P, \emptyset)$ is always defined. Moreover, if the function call is initiated with an empty set for R, all the recursive calls are evaluated to a disk and the results are always defined. This is due to three theorems introduced above. Notice that when this function is called with a set of points P and an empty set R, the following happens: The algorithm keeps checking if P is empty or if R contains exactly 3 points and they both fail every time. However, the algorithm removes a random point from P every time and at some point a recursive call is made with $P = \emptyset$ and $R = \emptyset$. Notice that the result of this call is not undefined. The algorithm returns an empty disk at this point: $D = \emptyset$. Furthermore, if the cardinality of the set R is exactly 3, then the algorithm computes a disk with the three points in R on its boundary. The question that can be raised here is: why is there no check to confirm that all the points in P are enclosed with the disk D? Let us assume that we perform such a test and discover that a point p is not covered by the disk, then we have to recursively call our function with $R \cup \{p\}$. This will result

Algorithm 9 Computes the smallest enclosing disk of points P with points R on the boundary of the disk.

```
Require: $P$: set of points to be covered by the disk
Require: $R$: set of points that should be on the boundary of the disk
  procedure B_MINIDISK($P$, $R$)
    if $P = \emptyset$ or $|R| = 3$ then
      $D := b\_md(\emptyset, R)$
    else
      $p \in P$ ; where $p$ is chosen randomly
      $D$ :=B_MINIDISK($P - \{p\}$, $R$)
      if $p \notin D$ then
        $D$ :=B_MINIDISK($P - \{p\}$, $R \cup \{p\}$)
      end if
    end if
    return $D$
  end procedure
```

in an undefined disk because the points in R are not co-circular. This will never happen if the function is initially called with a set of points P and an empty set for R. Therefore the test would not be required.

It must be clear that the validity of algorithm 9 can only be confirmed if the initial call is performed with $R = \emptyset$. It is crucial to ensure that this is done by another routine and therefore this algorithm is never called directly to ensure its validity. The algorithm that calls $b\_minidisk$ is very simple:

Algorithm 10 Computes the smallest enclosing disk that covers all the points in P.

```
Require: $P$: set of points to be covered by the disk
  procedure MINIDISK($P$)
    $D$ :=B_MINIDISK($P$, $\emptyset$)
    return $D$
  end procedure
```

We now show the running time analysis of this algorithm which is an instance of backward analysis that is commonly used for time and space analysis. We basically have to count the expected number of times that we run the test $p \notin D$. The actual running time is a constant number multiplied by this expected number. This is true as long as P is non-empty and if

$P = \emptyset$, then the total running time is $O(1)$. Let $t_j(n)$ denote the number of times we execute this test ($p \notin D$). In this notation, we let $n = |P|$ and $j = 3 - |R|$. Observe that $t_0(n) = 0$. This is because when $j = 0$, then $|R| = 3$ and therefore we do not execute the test $p \notin D$. Additionally, observe that $t_n(0) = 0$. This is due to the fact that this means $|P| = 0$ and therefore we do not execute the test. Let $j > 0$ and $n > 0$, we make one call to $b\_minidisk(P - \{p\}, R)$, then we run a test to check for $p \notin P$. Additionally, we may make another call to $b\_minidisk(P - \{p\}, R \cup \{p\})$ with a probability of at most $\frac{j}{n}$. The latter is due to the third property we introduced above. This will lead us to the following inequality:

$$t_j(n) \leq t_j(n-1) + 1 + (\frac{j}{n})t_{j-1}(n-1) \tag{5.12}$$

This evaluates to $t_1(n) \leq n$, $t_2(n) \leq 3n$ and $t_3(n) \leq 10n$. According to experiments in [33], this constant behind this linear complexity also turns out to be 10 for a set of points uniformly distributed in the plane. Finally, we conclude that the expected running time of this algorithm is $O(n)$.

5.6.3 Enclosing Disk Simplification

The randomized smallest enclosing disk algorithm provides a platform for the second solution to solve the trajectory simplification problem. This method is a modification of the polygonal chain simplification approach given by Bereg and Jiang [2]. This approach is also similar to the direct-link rule solution in the following way: The aim of the direct-link rule approach was to replace each partition of a trajectory with a shorter trajectory. The second solution follows along the same lines. We partition the trajectory using a special technique and then replace each partition with a shorter partition that spans over the same time interval. In order to find each partition, we combine the doubling search method and the randomized smallest enclosing disk algorithm.

Recall the partitioning method m-walk along a polygonal chain P. Our aim here is to construct a very similar approach with some modifications. Let us first define the $m\delta$-walk along a trajectory T:

Definition 5.2 (**$m\delta$-walk**). *Let $T = \{c_1, c_2, ..., c_n\}$ be a trajectory. The $m\delta$-walk along T partitions the trajectory into m disjoint, non-empty sub-trajectories $\{\mathbb{T}_i\}_{i=1..m}$ such that $\mathbb{T}_i = (c_{k_{i-1}+1}, ..., c_{k_i})$ and $0 = k_0 < k_1 < k_2 < ... < k_m = n$. Furthermore, the radius of the smallest enclosing disk D_i that covers all the points in the partition $\mathbb{T}_i$ is at most δ.*

Intuitively, the $m\delta$-walk along a trajectory partitions the trajectory into disjoint non-empty sub-trajectories such that the order of nodes are preserved. Moreover, the union of all these partitions contains all the spatio-temporal nodes in the original trajectory. The main difference between an m-walk and a $m\delta$-walk is that each partition in an $m\delta$-walk has an extra restriction: The radius of the smallest enclosing disk of the points in each partition is at most δ. Refer to Figure 2.13 for the polygonal chain example. The partitioning of a trajectory is also very similar.

The aim of our next algorithm is to partition an input trajectory as described above. Observe that such a trajectory always exists as long as δ is a non-negative number: Each node of the trajectory can be partitioned separately and therefore the radius of a smallest enclosing disk covering one point is zero. However, we are not looking for such a partitioning because it will be useless. The algorithm then makes an attempt to replace each partition with a shorter sub-trajectory. Bereg [2] does this by replacing each partition with a single point: the centre of the smallest enclosing disk that covers the points in that partition. However, it gets more complicated when it comes to trajectories because trajectories have timestamps. Let $\mathbb{T}_i$ be a partition such that the radius of the smallest enclosing disk covering the points in the partition is at most δ. In the case of trajectories, each partition $\mathbb{T}_i$ is replaced with another partition $\mathbb{T}_i{}'$ of at most the size of $\mathbb{T}_i$. Let o_i be the centre of the smallest enclosing disk that covers all the points in $\mathbb{T}_i$. Let $\mathbb{T}_i{}'$ be the simplified version of $\mathbb{T}_i$ which is computed as following:

- Case I: if $|\mathbb{T}_i| < 3$ then $\mathbb{T}_i{}' = \mathbb{T}_i$
- Case II: if $|\mathbb{T}_i| > 2$ then $\mathbb{T}_i{}' = (c_{i_1}, c_{i_2})$ where $c_{i_j} = (o_i, t_{i_j})$ and t_{i_1} and t_{i_2} are the smallest and the largest timestamps in partition $\mathbb{T}_i$ respectively.

Observe that each partition is a sub-trajectory and its replacement is also another sub-trajectory. Let $\mathbb{T}_i$ be a sub-trajectory obtained from an $m\delta$-walk. Furthermore, let D_i be the smallest enclosing disk that covers all the points in partition $\mathbb{T}_i$. Moreover, let o_i and r_i be the centre and the radius of this disk respectively. Observe that r_i is always less than or equal to an arbitrary constant which we called δ. Let us now analyze the trajectory distance between $\mathbb{T}_i$ and $\mathbb{T}_i{}'$. Notice that for those cases where $|\mathbb{T}_i| < 3$, the trajectory distance is zero: $\delta_T(\mathbb{T}_i, \mathbb{T}_i{}') = 0$. However, for cases where $|\mathbb{T}_i| > 2$, the trajectory distance can be greater than zero. Observe that $\mathbb{T}_i$ and $\mathbb{T}_i{}'$ are defined on the same time interval. Moreover, all the points in $\mathbb{T}_i$ are covered in D_i and $\mathbb{T}_i{}'$ remains stationary at the centre of D_i which we called o_i. The maximum distance of any point covered in D_i to the centre of the D_i is r_i. Hence, the trajectory distance is at most r_i: $\delta_T(\mathbb{T}_i, \mathbb{T}_i{}') \leq r_i$. Since r_i is always less than or equal to δ, we conclude that $\delta_T(\mathbb{T}_i, \mathbb{T}_i{}') \leq \delta$.

Once a trajectory is partitioned and each partition is simplified, we simply link the simplified partitions in their respective order. In other words, each $\mathbb{T}_i$ is replaced with its respective

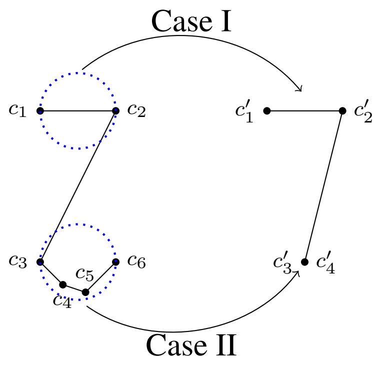

Figure 5.4: In the figure above, $T = \{c_1, c_2, ..., c_6\}$ is simplied to $T' = \{c'_1, c'_2, c'_3, c'_4\}$. T is divided into two disjoint partitions. The first partition has only two nodes. Thus, Case (I) applies and it remains the same. In other words, $c'_1 = c_1$ and $c'_2 = c_2$. However, the second partition has 4 points and therefore Case (II) is applied. Both of the nodes c'_3 and c'_4 are positined at the same point which is the center of the smallest enclosing disk of points in the second partition. However, the timestamp of c'_3 is equal to the timestamp of c_3 and the timestamp of c'_4 is equal to the timestamp of c_6. This is because these are the smallest and the largest timestamps of the nodes in this partition.

computed simplified sub-trajectory: $\mathbb{T}_i{}'$. In order to illustrate this mathematically, consider the following: Let $ts(c)$ denote the timestamp of the spatio-temporal node c. Additionally, let $A = (a_1, a_2, ..., a_n)$ and $B = (b_1, b_2, ..., b_m)$ be two trajectories such that $ts(a_n) < ts(b_1)$. Furthermore, let $A \circ B$ denote a trajectory that is the concatenation of the two trajectories A and B: Mathematically, $A \circ B = (a_1, a_2, ..., a_n, b_1, b_2, ..., b_m)$. Consider T as a trajectory that is to be simplified. Moreover, an $m\delta$-walk along T is performed to obtain a partitioning of T: $\{\mathbb{T}_i\}_{i=1..m}$. Each partition $\mathbb{T}_i$ is simplified to obtain $\mathbb{T}_i{}'$. The simplified version of T, namely T' is the result of concatenating all the simplified partitions:

$$T' = \mathbb{T}_1{}' \circ \mathbb{T}_2{}' \circ ... \circ \mathbb{T}_m{}' \tag{5.13}$$

Recall that the discrete trajectory distance is defined as following:

$$\delta_{dT}(P, Q) = \max(\max_{1 \le i \le n}(d_{PQ}(ts(p_i))), \max_{1 \le i \le m}(d_{PQ}(ts(q_i)))) \tag{5.14}$$

where $d_{PQ}(t)$ is the Time-Distance function of P and Q at time t. This definition was originally given in Chapter 4 (Definition 4.4).

Consider T and its simplified version T' and their partitions: $\mathbb{T}_{i=1..m}$ and $\mathbb{T}'_{i=1..m}$. Then the discrete trajectory distance for T and T' can be written as following:

$$\delta_{dT}(T, T') = \max_{i=1..m} (\delta_{dT}(\mathbb{T}_i, \mathbb{T}'_i)) \tag{5.15}$$

We already showed that after partitioning a trajectory T using an $m\delta$-walk and computing simplified partitions, the following holds: $\delta_T(\mathbb{T}_i, \mathbb{T}_i{}') \leq \delta$. Due to the equality above, we conclude that $\delta_{dT}(T, T') \leq \delta$.

Let us now look at the algorithm. We already mentioned that the final algorithm is a combination of doubling search method and the randomized smallest enclosing disk algorithm. For the doubling search method, we know that we need a concrete rule. We define the enclosing disk rule function that corresponds to R_{disk}:

Definition 5.3 (Enclosing disk rule function). G*iven a trajectory* $T = \{c_1, c_2, ..., c_n\}$, *an index* i *and a threshold value* $\epsilon \geq 0$, *the enclosing disk rule function as following:*

$$f_{disk}(T, i) = \begin{cases} true & \textit{if } radius(D_i) \leq \epsilon \\ false & \textit{if } radius(D_i) > \epsilon \end{cases}$$

where D_i *denotes the smallest enclosing disk that covers all the points in the first* i *nodes of* T *and* $radius(D)$ *denotes the radius of the disk* D.

The algorithm that evaluates the enclosing disk rule function is very simple to design. It basically has to call the function $minidisk$ with the first i nodes of T and check if the computed disk has a radius of at most ϵ. This function can be used in the doubling search algorithm as the ruling function. The optimal index set returned by the doubling search method is then used to simplify the trajectory as described above. Notice that $minidisk$ runs in expected linear time. Thus, the running time of this algorithm is expected $O(n \log n)$. Note that an actual linear running time algorithm exists to compute the smallest enclosing disk covering a set of points [24]. Hence, a worst-case $O(n \log n)$ algorithm is possible but in practise, we choose the randomized approach because it is simple and intuitive.

5.7 Conclusion

We have proposed two solutions to simplify a trajectory based on a threshold level δ. We will point out the advantages and disadvantages of each method and propose a few ways to test and compare them in practice.

Let us first start with the direct link simplification. The main disadvantage is that the direct link rule does not follow properties 2 and 3 of the doubling search method. This makes it hard to determine the final solution of an input trajectory. On the other hand, the direct link method is extremely advantageous in cases where an input trajectory rarely changes its direction or speed. In such cases, it ensures that the intermediate nodes on a straight line will be removed. Recall our main simplification goal: Convert an input trajectory to a simplified trajectory with as few nodes as possible. In fact, most trajectories rarely change direction or distance when you look at a large trajectory dataset. This method seems to be very promising in such cases.

The enclosing disk simplification method is exactly the opposite of the direct link simplification method. In fact, it is easy to determine the simplified version of an input trajectory. This is because the enclosing disk rule obeys all the 3 properties defined in the doubling search method. Consider an object moving through space on a straight line at a constant speed. The enclosing disk method is unable to remove all the intermediate nodes in this cases. This method works in batches. Its partitioning model simply is not aware of straight lines. In fact, straight lines may be the worst case scenario for this method. This method can perform better when a trajectory changes direction frequently.

In Chapter 7, we will execute both of these simplification methods and compare them in:

1. ***Performance:*** the amount of time it takes each method to simplify a large dataset of trajectories.

2. ***Error Distribution:*** This is a distribution of trajectory distances between input trajectories and their corresponding simplified version.

3. ***Size Ratio Distribution:*** This is a distribution of ratio between the size of the input trajectories and their simplified versions.

Performance is easy to analyze. The method that takes less time to finish wins the performance metric.

One goal of simplification of a trajectory is to reduce the number of nodes in an input trajectory. But at the same time, we would like to minimize differences between the original input trajectory and the simplified trajectory. Thus, we introduce the δ factor as a threshold. However, we can still analyze the differences caused by each method. The error distribution is the distribution of distances where each distance is equal to $\delta_{dF}(T, T')$ where T is an input trajectory and T' is the simplified version of T. In this case, we would like to see minimal change. Thus, we would like the distribution to have a large population of smaller distances (closer to 0).

Finally, the ultimate goal of simplification is to have as few nodes as possible in the simplified version of a trajectory. In order to analyze this effect, we will compute the size ratio of simplified trajectories over the size of the actual input trajectory. Given a trajectory T of size n, we compute its simplified trajectory T' of size m. Thus, the size ratio will be equal to $\frac{m}{n}$. Since m is at most equal n, this ratio will at most be equal 1. Moreover, m and n are both positive integers and therefore, this ratio is always a positive real number. In other words, this distribution will be defined in the range of values between 0 to 1. Observe that values closer to 0 indicate a better simplification. On the other hand, values closer to 1 indicate that the simplification had little affect on the size of the trajectory.

Chapter 6

Clustering

6.1 Introduction

Clustering is the most common unsupervised learning method in pattern recognition. It is basically the task of grouping or ”clustering” a set of objects such that similar objects reside in the same group together. Clustering algorithms have gained a lot of attention among researchers and therefore a large number of clustering algorithms have evolved. In order to be successful with clustering a set of objects, first we need to understand the nature of the objects we need to cluster. We need to examine their properties and understand how these objects are inputted to an algorithm. For instance, one may be interested to know whether the data that needs to be clustered is inputted to the algorithm incrementally. This is an interesting problem when data is collected as clusters are evolving. On the other hand, the data-set can be present as a whole prior to the execution of the clustering algorithm. Additionally, the properties of the objects that need to be clustered are very important. These properties can give rise to a number of important questions such as if the objects are vector-based or if the objects are metric based. Moreover, one needs to know if objects can be transformed from one form to another so that further analysis can be done on them. One of the most important aspects of clustering is a way of comparing the objects which is widely know as a distance function.

In this chapter, we will explore clustering algorithms and make an attempt to show which clustering algorithm is reasonable for trajectory clustering. We first start off with some prerequisites that many clustering algorithms require and show that these requirements are met by the trajectory distance function. Next, we move onto a traditional hierarchical clustering (HAC) algorithm and we will show how HAC attacks the problem. HAC is simple to understand and it may present us with some insight as to what we should expect from a general clustering algorithm. However, HAC is not an efficient algorithm and it requires the entire data-set in advance and therefore it may not be a good candidate.

Once we have some insight of a general clustering algorithm, we may move onto a more complex algorithm called the BIRCH algorithm [34]. BIRCH is now a class of algorithms and other variations of it have evolved over the past decade. It is an efficient and incremental algorithm. However, we will see its requirements on the input data-set are too strong. As a result, a variation of BIRCH known as BUBBLE [12] is studied that has very limited requirements.

After exploring BUBBLE, a variation of it which improves its performance known as BUBBLE-FM is shown in this chapter. However, BUBBLE-FM depends on another algorithm called FastMap. Hence, prior to explaining the workings of BUBBLE-FM [12], we will cover the details of FastMap. BUBBLE-FM turns out to be an efficient and incremental algorithm and therefore, it is the algorithm that we use for clustering trajectories.

6.2 Clustering Prerequisites

Clustering algorithms are generally designed to be able to target a large class of objects. However, some algorithms have specific assumptions on the type of objects that are to be clustered. The most general assumption is a mathematical definition for the objects so they can be stored in memory or on a storage device using a data structure. Perhaps the second most widely accepted requirement is a distance function so the algorithm would be able to compute the distance between the objects of interest. Some clustering algorithms require a vector-based representation of the objects so each object can be represented as a point in a k-dimensional space and normally the euclidean distance between these points is used as the distance function between the objects. Observe that such assumptions can be very strong in real world problems. We know that we have a mathematical representation of our trajectories. We have also discussed the distance function between the trajectories and we also know that the trajectory distance function follows the metric space rules: it is symmetrical and it follows the triangulation property. HAC (Hierarchical Agglomerative Clustering) algorithm only requires a distance function between the objects. It is the simplest clustering algorithm covered in this chapter. Moreover, HAC can cluster any set of objects as long as a distance function is mathematically defined to compare two objects. BIRCH algorithm [34] requires a vector-based representation of the objects and it uses the euclidean distance to compute distances between the objects. BUBBLE [12] which is a variation of BIRCH has a weaker assumption in the sense that the objects should only provide a metric-based distance function. BUBBLE removes the requirement that enforces objects to be defined as points in a k-dimensional space.

Moreover, BUBBLE-FM [12] uses an algorithm called FastMap [10] to map some of the objects into a k-dimensional space. This speeds up the object distance computation and therefore improves the running time of the clustering algorithm in practice. Furthermore, BUBBLE-FM imposes the same requirements on the objects and the distance function as the BUBBLE algorithm.

6.3 Hierarchical Agglomerative Clustering

Some clustering algorithms such as k-means clustering [23] require a constant value (k as input) for the number of clusters prior to execution. The main issue with such algorithms is the unknown nature of data. In a real world problem, one normally does not expect to have a precise knowledge of how the data is spread out across the space in advance. Therefore, it is a very strong assumption to expect the number of clusters in advance. In contrast, hierarchical clustering algorithms [13] try to tackle these issues by introducing a different view of how clusters can be constructed. Such algorithms usually require a measure of dissimilarity among clusters.

As the name suggests, hierarchical clustering algorithms produce a hierarchical representation of the data. Such hierarchical data representations are normally stored in a tree data structure. Each node of the tree is considered to be a cluster. The root of the tree is at the highest level and it contains all the elements. The root of the tree can be viewed as a single cluster. Each internal node of the tree contains children and each child can be interpreted as a single cluster. Assuming that the entire data-set is given in advance, there are two basic strategies:

- Agglomerative (bottom-up)
- Divisive (top-down)

The agglomerative approach starts at the bottom of the tree. Each object is inserted to a leaf node representing a single cluster containing a single object. The algorithm proceeds by merging similar object by constructing internal nodes containing several nodes from the previous level. This is done recursively until there is a single node which defines the root of the tree.

The divisive approach works is opposite direction. Unlike the agglomerative approach, the divisive approach starts off by creating the root of the tree. The algorithm continues by dividing the entire data-set into two subsets of similar objects. This approach is also recursive. The recursion continues until the division process reaches a subset with a single node.

In this section, we focus on HAC (Hierarchical Agglomerative Clustering) as an example of the hierarchical clustering algorithm. The main idea is to give some insight of how such algorithms are designed. This is also beneficial because the reader will have an idea of what these tree data structures may look like.

Assuming that we are given a data-set with n objects to cluster. HAC starts by creating a cluster for each object and then it performs $n-1$ merges. At each step, the two most similar clusters are merged together. These merges form new internal nodes of the tree containing more objects. Notice that when two nodes are being merged together, the algorithm needs to have a dissimilarity measurement between clusters. This means that the distance function is not enough. However, the cluster dissimilarity measure is derived from the provided distance function. HAC clustering may use one of the following common flavours of dissimilarity measurements:

- Single linkage (SL)
- Complete linkage (CL)
- Group average (GA)

The dissimilarity measurement is a derivation of distance function. Let G and H be two clusters, the Single linkage measurement is the least distance between any two objects, $g \in G$ and $h \in H$:

$$d_{SL}(G,H) = \min_{\substack{g \in G \\ h \in H}} d(g,h) \tag{6.1}$$

The complete linkage is the opposite of Single linkage: CL is the maximum distance between any two objects, $g \in G$ and $h \in H$:

$$d_{CL}(G,H) = \max_{\substack{g \in G \\ h \in H}} d(g,h) \tag{6.2}$$

Finally, Group average uses the average dissimilarity between groups:

$$d_{GA}(G,H) = \frac{1}{|G| \times |H|} \sum_{g \in G} \sum_{h \in H} d(g,h) \tag{6.3}$$

SL dissimilarity measure is known to be a local measure: It mainly focuses on the area where the two clusters are closest to each other. In other words, we pay no attention to the rest of the elements. We decide based on the closest elements. The more distant parts of the clusters are not taken into account. This may result in very scattered clusters.

On the other hand, CL is a non-local dissimilarity measurement: It focuses on the most distant elements. This results in very compact clusters. The issue with CL is its sensitivity to outliers.

This brings us to group average. The GA dissimilarity measure tries to address these issues. It solves the problem by averaging all the distances between the two clusters. This being said, CL and SL are sometimes better options for some specific data-sets.

Notice that regardless of which cluster dissimilarity measure is chosen, the final data structure is a binary tree. Furthermore, the children of the root represent the two most distant clusters. In fact, at higher levels of the tree, the distance between clusters are larger compared to the lower levels of the tree. Let ϵ be a defined clustering threshold. We traverse down the tree and at each node O with children C_1, C_2. Moreover, we check if $d(C_1, C_2)$ is at most ϵ. If so, we consider the sub-tree rooted at O as a whole cluster with respect to ϵ. This means the number of clusters can be anything between 1 to n but we do not have to predetermine that. Hierarchical clustering is a good method to cluster trajectories because we will not know the nature of our trajectories. However HAC has two major flaws when it comes to clustering trajectories. The most important issue with HAC is that it is not incremental. In other words, it requires all the data-set prior to clustering. This is problematic. Recall that the aim of our clustering system was to design an unsupervised real-time learning environment. Furthermore, the system must be able to maintain clusters of trajectories as they are captured. For instance, consider a traffic camera that synchronizes traffic lights. For a robust synchronization, a real time clustering algorithm is required and HAC does not provide that. The second issue with HAC clustering algorithm is its run-time performance. The best design of HAC clustering algorithm requires $O(n^2)$ computations of the distance function where n is the number of objects. This is quite expensive given the computation cost of trajectory distance function. In the next section we will look at BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) which is an incremental hierarchical clustering algorithm.

6.4 BIRCH Clustering

BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) [34] was the first incremental clustering algorithm handling noise effectively. BIRCH was introduced to handle multi-dimensional data-sets but soon after its publication, a whole class of BIRCH variations were introduced.

Even though trajectory objects are not defined based on k-dimensional vectors, BIRCH is a good place to start. This is because a large number of clustering algorithms evolved from

BIRCH and some of them remove this constraint. Additionally, we can find algorithms that map metric-based objects into points in k-dimensional spaces such as FastMap [10]. Moreover, BIRCH can be very beneficial to our system because it was exclusively designed to deal with large data-sets that are incrementally maintained. This is exactly the case for our trajectory clusters. We wish to maintain the trajectory clusters incrementally as they are discovered. BIRCH is advantageous over the older clustering algorithms such as probability-based clustering [6] or distant-based clustering. Such algorithms do not adequately consider large data-sets. BIRCH has a hierarchical structure and therefore it maintains a tree data structure known as CF-Tree. CF stands for Cluster Feature. We need to define a few terms before explaining the workings of CF-Trees and their maintenance:

Let $X = \{\vec{X_1}, \vec{X_2}, ..., \vec{X_n}\}$ be a cluster with n d-dimensional data points:

- The centroid $\vec{X}$ is defined as:

$$\vec{X} = \frac{\sum_{i=1}^{n} \vec{X_i}}{n} \tag{6.4}$$

- The radius, R, which is the average distance from member points to the centroid is defined as:

$$R = \sqrt{\frac{\sum_{i=1}^{n} (\vec{X_i} - \vec{X})^2}{n}} \tag{6.5}$$

- The diameter D is the average pairwise distance within a cluster and it is defined as:

$$D = \sqrt{\frac{\sum_{i=1}^{n} \sum_{j=1}^{n} (\vec{X_i} - \vec{X_j})^2}{n(n-1)}} \tag{6.6}$$

Notice that these measurements are defined within a single cluster. Moreover, radius or diameter are good candidates for measuring the tightness of a cluster. However, BIRCH needs some measurements between clusters as well. In fact, BIRCH introduces five different types of such measurements:

Let X_1 and X_2 be two different clusters:

- The centroid Euclidean distance, $D0$, is defined as:

$$D0 = \sqrt{(\vec{X_1} - \vec{X_2})^2} \tag{6.7}$$

- The centroid Manhattan distance, $D1$, is defined as:

$$D1 = \left|\vec{X_1} - \vec{X_2}\right| = \sum_{i=1}^{d} \left|\vec{X_1}^{(i)} - \vec{X_2}^{(i)}\right| \tag{6.8}$$

For the next three measurements, we need to define the indices somewhat differently: Given n_1 d-dimensional data points in a cluster $\{\vec{X_i}\}$ where $i = 1, 2, ..., n_1$ and n_2 data points in a second cluster $\{\vec{X_j}\}$ where $j = n_1 + 1, n_1 + 2, ..., n_1 + n_2$:

- Average inter-cluster distance, $D2$:

$$D2 = \sqrt{\frac{\sum_{i=1}^{n_1} \sum_{j=n_1+1}^{n_1+n_2} (\vec{X_i} - \vec{X_j})^2}{n_1 n_2}} \tag{6.9}$$

- Average intra-cluster distance, $D3$:

$$D3 = \sqrt{\frac{\sum_{i=1}^{n_1+n_2} \sum_{j=1}^{n_1+n_2} (\vec{X_i} - \vec{X_j})^2}{(n_1 + n_2)(n_1 + n_2 - 1)}} \tag{6.10}$$

- Variance increase distance, $D4$:

$$\begin{aligned} D4 = & \sum_{k=1}^{n_1+n_2} \left(\vec{X_k} - \frac{\sum_{i=1}^{n_1+n_2} \vec{X_i}}{n_1 + n_2} \right)^2 - \sum_{i=1}^{n_1} \left(\vec{X_i} - \frac{\sum_{l=1}^{n_1} \vec{X_l}}{n_1} \right)^2 \\ & - \sum_{j=n_1+1}^{n_1+n_2} \left(\vec{X_j} - \frac{\sum_{l=n_1+1}^{n_1+n_2} \vec{X_l}}{n_2} \right)^2 \end{aligned}$$

We can now cover how BIRCH works and how CF-Trees are maintained. BIRCH clustering relies on a data structure called a CF-Tree. Each internal node of the tree stores pointers to its children. Additionally, for each child, a triple summarization of the data points rooted at that child is also stored. This summary is called a ”Cluster Feature” or in short we call them CF. This explains the name of this data structure: CF-Tree.

Given N d-dimensional data points in a cluster: $\{\vec{X_1}, \vec{X_2}, ..., \vec{X_N}\}$, the cluster feature CF is defined as a triple: CF= $(N, \vec{LS}, SS)$, where N is the number of data points in that cluster, $\vec{LS}$ is the linear sum of all the data points: $\sum_{i=1}^{N} \vec{X_i}$ and SS is the square sum of all the data points: $\sum_{i=1}^{N} \vec{X_i}^2$.

The cluster features are at the core of CF trees and they empower BIRCH for two main reasons. Cluster features can be efficiently maintained. Consider when two clusters C_1 and C_2 with cluster features CF_1 and CF_2 are being merged together. The resulting CF can simply be computed as follows:

$$CF = CF_1 + CF_2 = (N_1 + N_2, \vec{LS_1} + \vec{LS_2}, SS_1 + SS_2) \tag{6.11}$$

In other words, cluster features are additive and the proof of this fact consists of very intuitive algebraic rules. The other important benefit of using cluster features is that it enables us to compute the tightness measures of a cluster: $\vec{X}, R, D$. In fact, not only the tightness measures but also in-between cluster measures can also be computed very efficiently given cluster features of two cluster: $D0, D1, D2, D3$ and $D4$.

Cluster feature representation provides us with a base to introduce the structure of CF-Trees which is a height-balanced tree with two main parameters: branching factor B and tightness threshold T. Moreover, each internal node of a CF tree consists of an array of the form $[CF_i, child_i]$ for each child of that node. The size of this array is at most B (also known as branching factor). This value limits the number of children that is allowed for each internal node. $child_i$ is a pointer to the i^{th} child of the node and CF_i is its cluster feature. Furthermore, each leaf node of the tree consists of L entries (each associated with a cluster). It is also worth noting that in some implementations, L can be chosen to be the same as B. Each entry in a leaf is associated with a cluster. Each leaf has an array of the form $[CF_i]$ which is the cluster feature of the i^{th} cluster in that leaf node. Additionally, each cluster of the leaf node must satisfy the threshold requirement of the tree with respect to the value T. This requirement can be optionally enforced on the radius of the clusters or the diameter of the clusters (this is left for implementation).

Internal nodes in a CF-Tree are used to guide a new cluster to the correct leaf node. Once the cluster reaches a leaf node, the cluster features of the leaf node will guide the cluster to the right internal cluster. If no appropriate cluster is found, a new insertion forms a new cluster in the leaf node. We will discuss the details of insertion further in this section. Notice that the data points are not stored in the CF-Tree directly. The data points are inserted into a database on the disk along with a cluster ID. This enables us to make use of disk space which is normally a large storage area. Therefore, a large number of data points can be maintained. Also observe that we only need the cluster features to perform insertions. The absence of data points from the CF-Tree does not impose any limitations when it comes to insertions.

In order to insert new objects incrementally to a CF-Tree, the object must be inserted into a cluster containing a single object and the cluster feature must be precomputed which is very intuitive. Next, the first phase of insertion starts by identifying the appropriate leaf. The new insertion recursively descends the CF-Tree by choosing the closest child node at each level of the tree using a chosen distance metric function: $D0, D1, D2, D3$ or $D4$. It is optional to choose a distance function. Once the appropriate leaf node is found, the second phase of the insertion start to modify the leaf. In this phase, the closest leaf entry is chosen based on the chosen distance function, next we check if merging the closest cluster violates the threshold condition. In the case where the insertion causes no violation this merging takes place and we are done with the second phase of the insertion. On the other hand, if the threshold condition is not satisfied, a new entry is added to the leaf containing the insertion. If the number of clusters in the leaf is at most equal to L then the modification phase is done. However, if the new cluster causes the number of clusters to increase beyond the limit L, then the leaf node splits into two separate leaf nodes. The separation is done by choosing the most distant clusters as seeds and redistributing the rest of the clusters based on the closest criteria. The third phase of insertion which is to modify the path to the leaf is very straight forward if the leaf node did not split. In this case, simply the cluster feature of each node is updated by adding the cluster feature of the newly inserted element. In case there was a split at the leaf node, the parent requires to insert a new entry describing the newly created leaf node. If the parent has enough room to create a new entry then the new entry is simply inserted and the cluster features are updated and we recursively do the update for each parent node up to the root of the tree. However, if there is not enough room (the number of entries is more than B) then we need to split the internal node similar to the leaf node and redistribute the node into two internal nodes. This can be done recursively up to the root and at each level the cluster features need to be updated. If the root of the tree is split then the height of the tree is incremented by one.

The original BIRCH clustering algorithm has another phase of insertion. Additionally, in order to improve the clustering quality the original algorithm goes through a process called ”global clustering”. The purpose of this section was to give some background about BIRCH clustering algorithm. This provides a platform for us to introduce the BUBBLE clustering algorithm. This is due to similarities between these two algorithms. In fact, BUBBLE has evolved out of BIRCH. Further processing in the original BIRCH is omitted here.

6.5 BUBBLE Clustering for Metric Objects

In the previous sections, we have learned about hierarchical clustering algorithms. We also have seen two very different flavours of such algorithms. BIRCH is quite complex and it only deals with multi-dimensional data points that can be represented in d-dimensional vectors. Trajectory data is not vector based. It rather has its own definition. It may be possible to map trajectories into a d-dimensional space but it may be a very complex process. Trajectories are defined and a distance function is introduced to compare them. Hence, the BIRCH clustering algorithm cannot be directly applied to trajectory data-sets. On the other hand, BUBBLE [12] is a variation of BIRCH that has a weaker assumption on the data. In fact, the only assumption that BUBBLE makes is on the distance function. BUBBLE requires the distance function to satisfy the triangle inequality property. Chapter 4 covered the trajectory distance function and showed that it follows the triangulation inequality. Hence, it is a good candidate for us. This is because BUBBLE shares all the nice properties that BIRCH provides and we need for our clustering. It also removes the one requirement that our trajectories do not meet (a vector-based definition). BUBBLE is also incremental and hierarchical. This section covers the BUBBLE clustering algorithm.

The main challenge for BUBBLE is to define similar cluster tightness and inter-cluster dissimilarity measurements. Additionally, BIRCH cluster features do not apply here. This is due to the abstract nature of the data-set. BUBBLE attacks the latter problem by defining its own version of cluster features. These new cluster features are called ”general cluster features”. This is because of the fact that these features are defined on more general domain of data-sets. In other words, they are not only defined on d-dimensional vectors.

BUBBLE clustering algorithm works similar to BIRCH in the sense that it sequentially reads objects and inserts them into a tree. Hence, it enables us to perform clustering in real-time while the data is discovered. BUBBLE clusters are maintained incrementally. In order to be consistent, in this section we refer to cluster features and general cluster features interchangeably. Moreover, the tree maintained by BUBBLE is also called a CF-Tree. BUBBLE limits the number of nodes in a CF-Tree by a constant value M which can be chosen prior to its execution. Similar to BIRCH, BUBBLE also has a branching factor B. Additionally, the number of entries in the leaf nodes is also bounded by B. In other words, BUBBLE chooses the value of L to be equal to B. The form of entries in leaves and internal nodes of the tree is the same as BIRCH but the cluster features differ. Furthermore, each cluster at the leaf level must satisfy the threshold requirement T. This is to have control over the tightness or quality of the clusters. The purpose of the internal nodes are to guide new insertions to the right leaf node. Once the object reaches a leaf node, it is then inserted into the closest cluster in that leaf. If the tightness requirement is violated, then a new cluster is generated. If the number of clusters in a leaf is beyond the branching factor B, splitting and redistribution takes place. This may cause parent nodes to split up recursively as well. All of these operations are the same as BIRCH. When the number of nodes in the tree is greater than M, the tightness threshold T is increased and the tree is reconstructed. This will lead to fewer clusters because close clusters are merged together. The reconstruction of the tree is fairly simple. We start off with a new empty tree that has a higher tightness threshold T. Furthermore, all the clusters residing at the leaves of the previous tree are inserted into the new tree. The insertion procedure takes care of the merges. Finally, the new tree replaces the old tree and we are done.

Unlike BIRCH, BUBBLE has two different types of insertion. Type one insertion is the insertion of a single object to the tree. Type two insertion is the insertion of a cluster into the tree. Type two insertion only occurs when rebuilding the tree. Moreover, cluster features at the leaf level are different than those of the internal nodes. Due to the unknown structure of objects, the centroid of a cluster is undefined. Hence, we use an actual object $\hat{O}$ from a set of objects O as a clusteroid:

Definition 6.1 (RowSum). L*et* $O = \{O_1, O_2, ..., O_n\}$ *be a set of objects with a metric distance function* $d : O \times O \to \mathcal{R}$. *The* $RowSum$ *of an object* $o \in O$ *is defined as* $RowSum(o) = \sum_{i=1}^{n} d^2(o, O_i)$.

Definition 6.2 (Clusteroid). T*he clusteroid* $\hat{O}$ *is defined as an object* $\hat{O} \in O$ *such that* $\forall o \in O : RowSum(\hat{O}) \leq RowSum(o)$.

The rest of the definitions here depend on a distance-preserving function f as a base. Kruskal [20] has shown that such a function always exists. The following Lemma defines this function:

Lemma 6.1 *Let* $O = \{O_1, O_2, ..., O_n\}$ *be a set of objects with a metric distance* $d : O \times O \to \mathcal{R}$. *There exists a positive integer* k, *where* $k < n$ *and a function* $f : O \to \mathcal{R}^k$ *such that* f *is an* $\mathcal{R}^k$*-distance preserving transformation.*

Informally, for every object $o \in O$ there exits a mapping $f(o) = \vec{o'} \in \mathcal{R}^k$. Additionally for any pair of objects $\{o_1, o_2\} \subseteq O$, the metric distance $d(o_1, o_2)$ is equal to the euclidean distance between $\vec{o_1}'$ and $\vec{o_2}'$. Observe the clusteroid $\hat{O}$ of a set of objects O whose image vector $f(\hat{O})$ is the closest point to the centroid of the set O. Let $\overline{O}$ be the centroid of $f(O)$:

$$\forall o \in O : \left| f(\hat{O}) - \overline{O} \right| \leq \left| f(o) - \overline{O} \right| \tag{6.12}$$

BUBBLE also has its interpretation of radius R which is used to do approximation and to evaluate the quality of a cluster. Observe that the original BIRCH algorithm uses the radius or the diameter to determine the tightness satisfaction of a cluster but BUBBLE uses radius of a cluster for other purposes. Furthermore, BUBBLE has its own translation of distance measures $D0$ and $D2$. The average inter-cluster distance $D2$ is used at the internal nodes to guide a new object to the appropriate leaf. On the other hand, $D0$ is used to determine the appropriate cluster for a new insertion at the leaf level. Here are the formal definitions of R, $D0$ and $D2$ for the BUBBLE algorithm:

Let $O = \{O_1, O_2, ..., O_n\}$ be a set of objects with a metric distance function $d : O \times O \rightarrow \mathcal{R}$. The radius of the set O is defined as:

$$R(O) = \sqrt{\frac{\sum_{i=1}^{n} d^2(O_i, \hat{O})}{n}} \tag{6.13}$$

Let $O_1 = \{O_{11}, O_{12}, ..., O_{1n_1}\}$ and $O_2 = \{O_{21}, O_{22}, ..., O_{2n_2}\}$ with a metric distance function $d : (O_1 \cup O_2) \times (O_1 \cup O_2) \rightarrow \mathcal{R}$, the clusteroid distance $D0$ is defined as:

$$D0(O_1, O_2) = d(\hat{O}_1, \hat{O}_2) \tag{6.14}$$

Additionally, the average inter-cluster distance $D2$ is defined as:

$$D2(O_1, O_2) = \sqrt{\frac{\sum_{i=1}^{n_1} \sum_{j=1}^{n_2} d^2(O_{1i}, O_{2j})}{n_1 n_2}} \tag{6.15}$$

We already mentioned that BUBBLE algorithm does not use the radius of a cluster to test for the tightness requirement of a cluster. BUBBLE uses $D0$ to find the closest cluster to an object at the leaf level. It also uses $D0$ to test for the threshold requirement. In other words, a new object O_{new} is only inserted into a cluster O only if $D0(O, \{O_{new}\}) \leq T$. Let us now examine the incremental maintenance of cluster features at the leaf level. It appears that the clusteroid of a cluster is always required for most of the calculations so finding the clusteroid is vital. In order to find the clusteroid, we need to update the $RowSum$ of all the elements in the cluster and choose the object with the smallest $RowSum$. However, recall that BIRCH was originally designed to handle large data-sets and not all the objects in a cluster may be present in main memory. We need a strategy to keep some of the objects of a cluster in memory and store the rest of them in a database. Let us first examine the "type one" insertions (insertion of a single object O_{new}). In order to solve this problem, we make some approximations. Let C be a cluster and $\hat{C}$ be the clusteroid of that cluster. We approximate that the centroid of $f(C)$ to be close to $f(\hat{C})$. We already showed that this is the best approximation of the centroid:

$$\overline{f(C)} = f(\hat{C}) \tag{6.16}$$

We keep a maximum of p objects inside a cluster. If the total number of objects in the cluster is at most p, then the $RowSum$ of all the objects and O_{new} is computed exactly. Once the number of objects increase beyond p, we only update the $RowSum$ of p objects in memory and approximate the $RowSum$ of the new object in the cluster using the following:

$$\begin{aligned} RowSum(O_{new}) &= \sum_{i=1}^{n} d^2(O_{new}, O_i) \\ &= \sum_{i=1}^{n} (f(O_{new}) - f(O_i))^2 \\ &\approx \sum_{i=1}^{n} (f(O_i) - \overline{f(C)})^2 + n(\overline{f(C)} - f(O_{new}))^2 \\ &= nr^2(C) + nd^2(\hat{C}, O_{new}) \end{aligned} \tag{6.17}$$

After updating the $RowSum$ of the objects and inserting O_{new}, if $|C| > p$ then the object with the largest $RowSum$ is delivered to the database and removed from memory. Finally, the object with the smallest $RowSum$ value is the new clusteroid. Experiments confirm this heuristic works well in practice [12].

Let us now examine type two insertions. Recall that once the tree has more than M nodes, the value T is increased and the clusters at all the leaf levels are inserted into a new empty tree. In this process, clusters may get merged together resulting in a smaller tree.

Let $C_1 = \{O_{11}, ..., O_{1n_1}\}$ and $C_2 = \{O_{21}, ..., O_{2n_2}\}$ be two clusters being merged and let f be the distance-preserving transformation of $C_1 \cup C_2$. Let C be the result of merging C_1 and C_2. Observe that the centroid $\overline{f(C)}$ is on the line joining the two centroids $\overline{f(C_1)}$ and $\overline{f(C_2)}$. The exact location of the centroid depends of the values of n_1 and n_2. The new clusteroid $\hat{C}$ must be the closest object to this centroid. If the two clusters have more than p elements, we will only take p elements from each cluster. We assume that $n_1 \approx n_2$ and therefore we take a maximum of p elements from the periphery of each cluster. In this process, the objects stored in the database are read first before the objects in memory. This is because that the new clusteroid will be farther distance away from the two clusteroids. Moreover, the objects in the database have a larger $RowSum$ and therefore, they are more distant from the two clusteroids. If any of the clusters have more than p elements in the database, then we choose p random points from the database representing that cluster. These points are removed from the database and they get replaced by points in memory. As these points are inserted into the new cluster, we also update the cluster's features. Once the two clusters are merged, one of the two cluster IDs is chosen. This cluster ID is updated in the database as well. In summary, we maintain the following information as summarized cluster feature of a leaf cluster (C):

- number of objects in C

- clusteroid of C
- a maximum of $2p$ actual objects
- $RowSum$ of each object in memory.
- radius of the cluster (this may be approximated)
- cluster ID

We have described how leaf nodes behave and how they are maintained. We now describe the way internal nodes are maintained. As mentioned before, internal nodes have a different representation compared to the leaf nodes. Each entry in an internal node maintains a sample set of the objects in the sub-tree rooted at that node. This is the generalized cluster feature representation at the internal nodes. Let NL be an internal node in the tree and NL_i be the i^{th} entry of NL, then $S(NL_i)$ is the sample set of the entry NL_i and $S(NL)$ is the union of all sample object at all entries of NL.

These sample objects are randomly selected from the corresponding sub-trees. Let NL be an internal node which has k children: $child_1, ..., child_k$. $S(NL_i)$ is selected from objects rooted at $child_i$. If $child_i$ is a leaf node, then the sample objects $S(NL_i)$ are chosen from the clusteroids of the clusters existing under sub-tree rooted at $child_i$. Additionally, if $child_i$ is an internal node then these objects are chosen from the sample set of that internal node: $S(child_i)$. The number of sample objects to be collected at any internal node is upper bounded by a constant called the sample size. Let SS denote the sample size. The following inequality holds for the number of samples for th i^{th} child:

$$|S(NL_i)| \leq \lfloor \frac{n_i \times SS}{\sum_{i=1}^{k} n_i} \rfloor \tag{6.18}$$

The tree evolves as new objects are inserted into it. The frequency of the sample set updates will affect the quality of summary representations at internal nodes. On the other hand, too many updates can be computationally expensive so we need to find a good balance. When a split occurs at $child_i$ of NL we update all the sample set of the entries of NL and then we update the entry that points to NL we recursively follow this to the root. Notice, we only update all the entries of a node only if one of its children split but when moving back up the tree to the root we only update the corresponding entries.

The distance measurement at the internal nodes differ from the one used for leaf nodes. We use the average inter-cluster distance $D2$ as a measuring distance when inserting new objects. Let C_{new} be a new cluster being inserted into the tree (for type one insertion with a new object

O_{new} , this will be a cluster with a single object: $C_{new} = \{O_{new}\}$). C_{new} will be guided down the i^{th} entry of the internal node for which $D2(C_{new}, S(NL_i))$ is the smallest value. Notice this is different than the leaf nodes where the distance measurement was $D0$.

This is alarming because $D2$ computation is very expensive. Distance functions can be computationally expensive. This is where the BUBBLE-FM [12] algorithm comes into play. BUBBLE-FM decreases the number of calls to the distance function. BUBBLE-FM depends on an algorithm called FastMap [10] that maps metric objects to a coordinate system. FastMap tries to to preserve the distances between objects in their corresponding points in the coordinate system. Next section of this chapter covers the FastMap algorithm.

6.6 FastMap Algorithm

FastMap [10] is an algorithm used by BUBBLE-FM to map metric objects into points in a k-dimensional coordinate space. FastMap runs in $O(n)$ time where n is the number of objects to be mapped. Once the mapping is done, it can be used to answer queries by example. That is, given a new object O_{new}, FastMap can approximately map the new object into the same k-dimensional space. Moreover, the distance between the map of O_{new} and the centroid or the other points in the set can be computed efficiently. Additionally, FastMap allows us to choose a constant value k for the number of space dimensions. The higher the value of k, the more precision the algorithm provides in practice. There are other alternatives to FastMap such as Multi-Dimensional Scaling (MDS) [31] but MDS is very slow and it is not a practical solution for indexing.

The goal of FastMap is to find n points in a k-dimensional space whose Euclidean distances will approximately match the distances of a given $n \times n$ distance matrix. FastMap has access to the distance matrix or a distance function as input. FastMap pretends that the input data is indeed in an unknown n-dimensional space and it attempts to project them onto a k-dimensional space. FastMap does this one dimension at a time. For each dimension, two objects O_a and O_b are chosen as pivot objects. For the first dimension, we consider a line going through these two points in the original n-dimensional space and the rest of the points are projected on that line using the cosine law. Imagine projecting a point O_i onto a line that goes through O_a and O_b. Consider, the triangle $O_aO_iO_b$, the cosine law gives the following:

$$d^2(O_b, O_i) = d^2(O_a, O_i) + d^2(O_a, O_b) - 2x_i d(a, b) \tag{6.19}$$

Equation 6.19 above follows from the Pythagorean theorem. Let E be the projection point of O_i on the line that passes through O_a and O_b. Equation 6.19 comes from combining the

Pythagorean theorem in the two triangles O_aEO_i and O_bEO_i.

In the equation above, d is the distance function between the objects. For now we consider the first dimension. Let us now reformulate the equation and solve it for x_i:

$$x_i = \frac{d^2(O_a, O_i) + d^2(O_a, O_b) - d^2(O_b, O_i)}{2d(a, b)} \tag{6.20}$$

We raise two questions: How can this be extended to more than one dimension? How are the pivot points selected? Let us first address the first question. Recall that FastMap pretends that the objects are already points in an n-dimensional space. Consider an $(n-1)$-dimensional hyperspace that is perpendicular to the line going through O_a and O_b. Now consider projections of our points from the n-dimensional space to this $(n-1)$-dimensional hyperspace. That is for each point O_i in the original n-dimensional space, we have a projection $O_i{}'$ in the $(n-1)$-dimensional hyperspace. The goal is to find a new distance function that accounts for distances between projected points and using this new distance function to find two new pivot points and compute new values for the second dimension. We repeat the same process for higher dimensions by decrementing n and k again. We now solve for a distance function that computes the distances between the projected points:

Lemma 6.2 *On the hyper-plane H, the Euclidean distance d' between the projections $O_i{}'$ and $O_j{}'$ can be computed from the original distance function:*

$$d'^2(O_i{}', O_j{}') = d^2(O_i, O_j) - (x_i, x_j)^2$$

The proof of Lemma 6.2 can be found in the original paper [10]. We now address the second question, how can the pivot points be chosen at each step? Observe that the goal is to find two points O_a and O_b such that the two points are as far apart as possible in the projected hyper-plane. However, in order to find the most distant points we need $O(n^2)$ computations of the distance function which is very inefficient. Instead of doing this, we take a random point O_r and we find the most distant point to O_r and we call this new point O_a then we find the most distant point to O_a and we call this second point O_b. Observe that it only takes linear time to find O_a and O_b and these points are relatively far away from each other. Algorithm 11 shows this in pseudocode. Now we can describe how the FastMap algorithm works. FastMap takes three elements as input: a set of objects to map O, a distance function d that obeys the triangulation property and a constant value k representing the number of dimensions. The algorithm then finds two pivot points far away from each other using algorithm 11 and projects all the points to the line going through the two points. It is important to note that the pivot points for each dimension are stored. The algorithm then formulates a new distance function d' from

Algorithm 11 Choose objects far away from each other from a set of metric objects O.

```
Require: $O$: set of metric objects.
Require: $d$: distance function to compute distances between objects
Ensure: Find objects $O_a$ and $O_b$ that are relatively far away.
  procedure CHOOSEDISTANTOBJECTS($O$, $d$)
      $O_r \leftarrow$ a random point in $O$
      $O_a \leftarrow$ most distant point to $O_r$(using a linear search)
      $O_b \leftarrow$ most distant point to $O_a$(using a linear search)
      return $\{O_a, O_b\}$
  end procedure
```

d with the help of Lemma 6.2. Furthermore, d' is used to compute distances between objects on a hyper-plane perpendicular to the line going through the two pivot points. Moreover, d' turns out to be a recursive function for further dimensions. Algorithm 12 illustrates these operations in pseudocode.

Notice that the algorithm stores the pivot positions at each level in a 2D array called PA. Once the algorithm terminates, an array is generated with all the final mappings (k-dimensional points). This array is called X in the algorithm. Additionally, $X[i, j]$ contains the value of the j^{th} dimension for the i^{th} object. Observe that the algorithm runs in time O(kn) and since k is a constant value, the running time ultimately can be translated to $O(n)$. Additionally, once the algorithm returns the centroid of all the points in array X can be computed in linear time. Moreover, if a new object O_{new} is to be queried then O_{new} can be mapped to a point p using the PA in (k). The distance between p and the computed centroid can be obtained in constant time.

6.7 BUBBLE-FM Clustering

BUBBLE-FM [12] is a variation of the BUBBLE algorithm that improves the performance of the insertion operation. This is vital to the clustering algorithm because insertion of single objects into the CF-Tree is the most frequent operation. Improving this single operation will have a crucial effect on the running time of the entire algorithm.

Recall that BUBBLE stores general cluster features for each child of the internal nodes. One the components of the general cluster features in an internal node was a collection of clusteroids rooted at the associate child. This collection is called a sample set and it consists of

Algorithm 12 Map metric objects O into a k-dimensional space such that distances determined by distance function d are preserved as well as possible

```
Require: $O$: set of metric objects.
Require: $d$: distance function to compute distances between objects
Ensure: $X$ contains the $k$-dimensional points
Ensure: $PA$ contains the pivot positions at each level
  procedure FASTMAP($k$, $d$, $O$)
      if $k \leq 0$ then
          return
      end if
      $\{O_a, O_b\} \leftarrow$ CHOOSEDISTANTOBJECTS($O$, $d$)
      $PA[1, k] \leftarrow a$
      $PA[2, k] \leftarrow b$
      if $d(O_a, O_b) = 0$ then
          $X[i, k] \leftarrow 0$ for all $i$ such that $1 \leq i \leq |O|$
      else
          for $O_i \in O$ do
              $X[i, k] \leftarrow \frac{d^2(O_a,O_i)+d^2(O_a,O_b)+d^2(O_b,O_i)}{2d(O_a,O_b)}$
          end for
      end if
      Formulate $d'$ based on Lemma 6.2
      FASTMAP($k - 1$, $d'$, $O$)
  end procedure
```

pointers to actual objects in memory. In the previous section, we saw how FastMap [10] can be used to map metric objects into points in a k-dimensional coordinate space. BUBBLE-FM uses FastMap at its internal nodes to improve the running time of the insertion operation.

Let NL be an internal node of the CF-Tree who has k children: $child_1, child_2, ..., child_k$. Furthermore, let NL_i be the entry associated with the i^{th} element of NL. We described that each NL_i has a sample set that is derived from its children which we call $S(NL_i)$. Moreover, let $S(NL)$ be the union of all the sample sets of the children of NL:

$$S(NL) = S(NL_1) \cup S(NL_2) \cup ... \cup S(NL_k) \tag{6.21}$$

BUBBLE-FM stores some additional information about these sample sets. When one of the sample sets has to be modified, the modification is done the same way as BUBBLE. However, after these modifications are carried out, the FastMap of $S(NL)$ is computed. This computation results in the pivot array (PA) and the mapped points array (X). BUBBLE-FM stores the pivot array at the node NL. Additionally, BUBBLE-FM uses the mapped points array to compute the centroids of each sample set. Each NL_i also stores the centroid computed that is associated to its mapped points.

These new set of information are really useful when guiding down a new object O_{new} down the tree for insertion. Recall that BUBBLE uses $D2$ to choose the right child to guide down the new object. $D2$ is the average inter-cluster distance and it requires us to execute the distance function for $|S(NL)|$ times at each internal node NL. It is easy to see that this is a very inefficient operation. This is especially true if the distance function is computationally expensive. With the new information stored at each internal node, let O_{new} be a new object to insert into the CF-Tree. BUBBLE-FM does not use $D2$ to find the right child at each level. In contrast, it will use the pivot array stored at the internal node to map O_{new} to a d-dimensional point. Observe that this operation only takes time proportional to the number of dimensions: $O(d)$. Once this mapping is done, we can compute its distance to each centroid stored at NL_i. Observe that this operation takes time proportional to the number of children stored at NL: $O(k)$. The child whose centroid is the closest to the mapped point is chosen. This reduces the number of computations to find the right child at each level from $|S(NL)|$ to only $k + d$ operations where k is the number of children and d is the number of dimensions used in the FastMap algorithm. Observe that k is bounded by the branching factor B in the tree which is normally chosen to be at most 7. Also d is the number of dimensions and for most forms of data, 5 dimensions are enough. This conclude that we only need a total of 12 operations at each level to choose the right child to push the new object O_{new} down the tree.

It is important to note that FastMap runs in linear time with respect to the number of input objects. We do not run FastMap very frequently. In fact, we only run FastMap when the sample sets need to be updated. This only occurs when a leaf node splits up and it causes a ripple effect towards the root of the tree.

Chapter 7

Results

7.1 Dataset

In this section, we give an overview about the dataset used for this thesis. The Geolife project [36, 35] collected 17621 trajectories over 5 years from April 2007 to August 2012. The project was supervised by Microsoft Research Asia and involved trajectories of 182 users. The data was recorded by different devices such as GPS loggers and GPS phones. The entire dataset is distributed over 30 cities in China and some cities in the USA and Europe. However, the majority of the data was collected in Beijing, China. Therefore, we decided to use the trajectories recorded in this specific region. The data is recorded based on decimal degrees in three dimensions of longitude, latitude and altitude. In order for the data to be valid for us, we ignore the altitude and we only use two dimensions. While parsing the dataset into trajectory arrays, we noticed that some trajectories have substantial delays between consecutive nodes. This is not desirable for us but we had a simple solution. We decided to introduce a minimum time threshold of 10 minutes between two direct nodes. If any trajectory has two nodes in a row that are more than 10 minutes apart, we split the trajectory. Furthermore, we are more interested in longer trajectories and therefore our system removes any trajectory with less than 240 nodes. The city of Beijing is defined in a region with longitudes in the ranging from 115.8 to 117.4 and latitude ranging from 39.4 to 40.8. Any trajectory containing nodes outside of this region is removed from the sample set. Furthermore, the data is present in a very short range due to the decimal degree unit. In order to represent the coordinates easier in the system, we scale the coordinates by multiplying each coordinate by 50000. This maps the points to a larger space that makes it easier for us to draw and analyze the curves. The total number of trajectories we obtained was 19973. The shortest trajectory has 240 nodes and the longest trajectory has 12488 nodes. Table 7.1 shows the frequency table with respect to the number of nodes in the trajectories. Table 7.2 shows the ranges of the scaled longitude and the latitude.

Node count range	**Number of trajectories**
$[240, 1248]$	16372
$[1249, 2497]$	2810
$[2498, 3746]$	542
$[3747, 4995]$	138
$[4996, 6244]$	64
$[6245, 7493]$	23
$[7494, 8742]$	11
$[8743, 9991]$	5
$[9992, 11240]$	3
$[11241, 12488]$	5

Table 7.1: Relationship between the node count range and the number of trajectory's in the dataset.

	Minimum	**Maximum**
Scaled Longitude	1178.89	76497.4
Scaled Latitude	453.262	69671.5

Table 7.2: After scaling the coordinates by multiplying them by 50000, the table above shows the minimum and maximum coordinates after scanning all the trajectories.

Figure 7.1 is a representation of all the trajectories in one image. After loading all the trajectories into our application, we sketched all trajectories into a single image and produced this image. Please note that this image is scaled to fit the page.

7.2 Linear vs Quadratic distance function algorithms

In Chapter 4, we illustrated two algorithms to compute the trajectory distance function. Algorithm 3 runs in quadratic time with respect to the number of nodes. This will clearly run much slower than the second algorithm which runs in linear time (Algorithm 4). Table 7.3 compares the running time performance of the two algorithms for multiple executions. We executed both

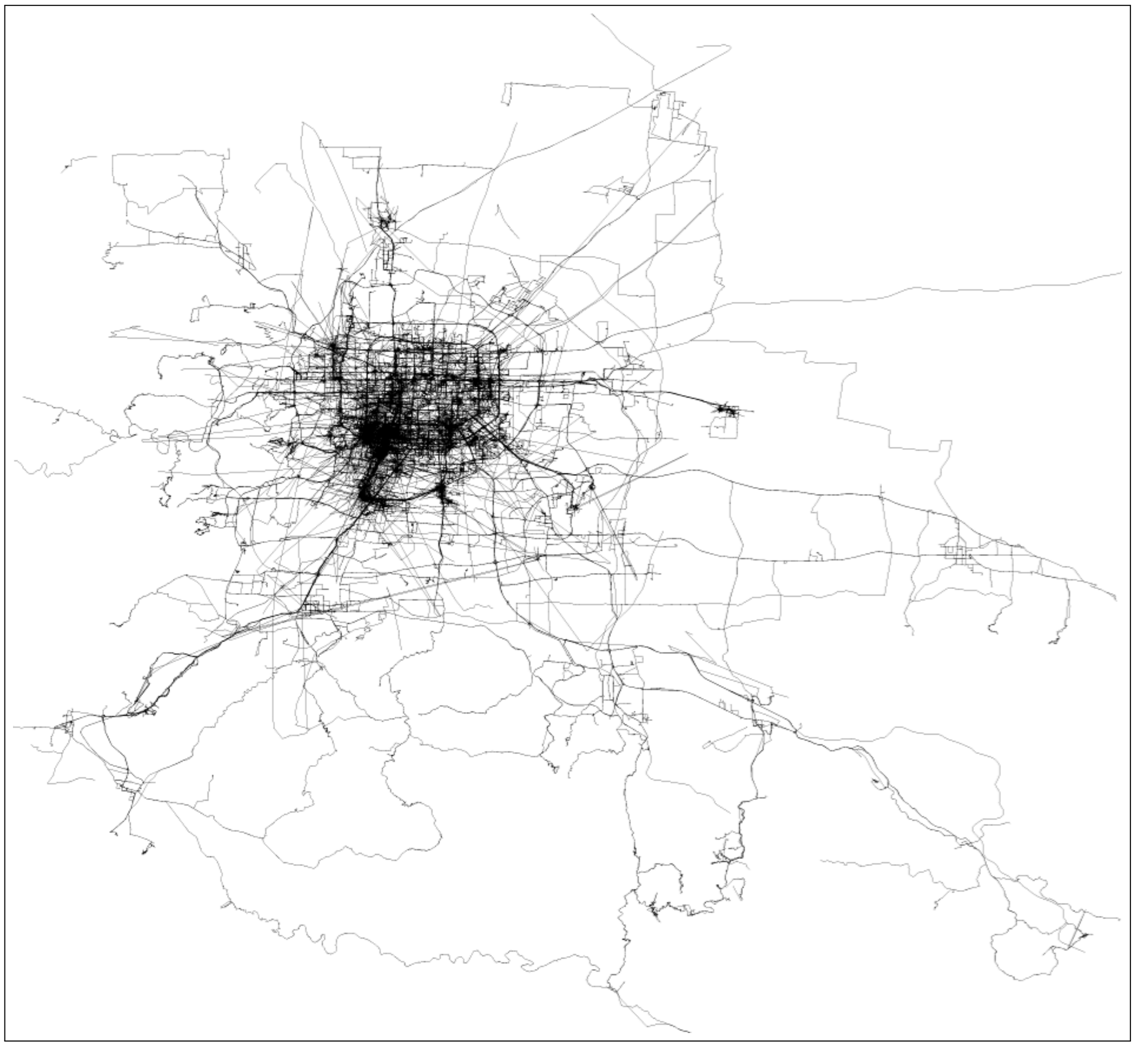

Figure 7.1: This figure is the result of sketching 19973 raw trajectories into a single image. These trajectories are used as a sample set for our tests.

of these algorithms with multiple random pairs of trajectories from the dataset and recorded their performance. The results show that the linear time algorithm computes the trajectory distance of a randomly chosen pair of trajectories picked from our dataset at 0.617 milliseconds on average. On the other hand, the quadratic algorithm does the same computation at 61.791 milliseconds on average. Thus, the quadratic algorithm is about 100 times slower than the linear algorithm on average for our data-set.

Number of trajectory pairs	Total running time (Linear Algorithm)	Total running time (Quadratic Algorithm)
652	$403ms$	$25,011ms$
68	$47ms$	$2,455ms$
1471	$912ms$	$55,367ms$
2314	$1,435ms$	$89,201ms$
547	$327ms$	$18,825ms$
981	$617ms$	$38,793ms$
1232	$775ms$	$50,341ms$
532	$333ms$	$21,551ms$
2358	$1,429ms$	$89,442ms$
2184	$1,335ms$	$79,433ms$

Table 7.3: The first column shows the total number of trajectory pairs used by the two algorithms. The second column shows the total running time of the linear algorithm to compute their distances. The third column shows the total running time of the quadratic algorithm to compute their distances.

7.3 Simplification Experiments

In this section, we will show some data points after applying each simplification method. We will show how the two trajectory simplification algorithms compare to each other.

7.3.1 Running-Time Performance

Let us first show some data points about the running time performance of the two simplification algorithms.

Algorithm	Running Time	Average
Enclosing Disk Simplification	$181,791ms$	$9.1ms$
Direct Link Simplification	$61,622ms$	$3.08ms$

Table 7.4: The results above shows the time it takes each algorithm to simplify our entire input dataset. The right-most column shows the average amount of time it takes to simplify a single trajectory.

This experiment reveals that the running time of the direct-link simplification method is about 3 times faster than the enclosing disk method. This is expected because the computation of the smallest enclosing disks is more expensive in practice. However, the simplification of a trajectory is only done once after the trajectory is detected in the system. On average, both algorithms run reasonably very fast for a single trajectory simplification. We need other metrics to measure their performance to identify the dominating algorithm.

7.3.2 Error Distribution

Error distribution is another metric to compare our trajectory distance algorithms. Let us first define the simplification error. Let S_ϵ be a simplification method that maps a trajectory T to a simplified trajectory T' with a threshold of ϵ. The simplification error, $e_{S_\epsilon T}$,is defined as the trajectory distance between the original trajectory T and the simplified trajectory T':

$$e_{S_\epsilon T} = \delta_T(T, S_\epsilon(T)) \tag{7.1}$$

Trajectory simplification methods loose some information about the trajectories which can be measured using the error defined above. The intention is to obtain a lower difference between the original trajectory and its corresponding simplified version.

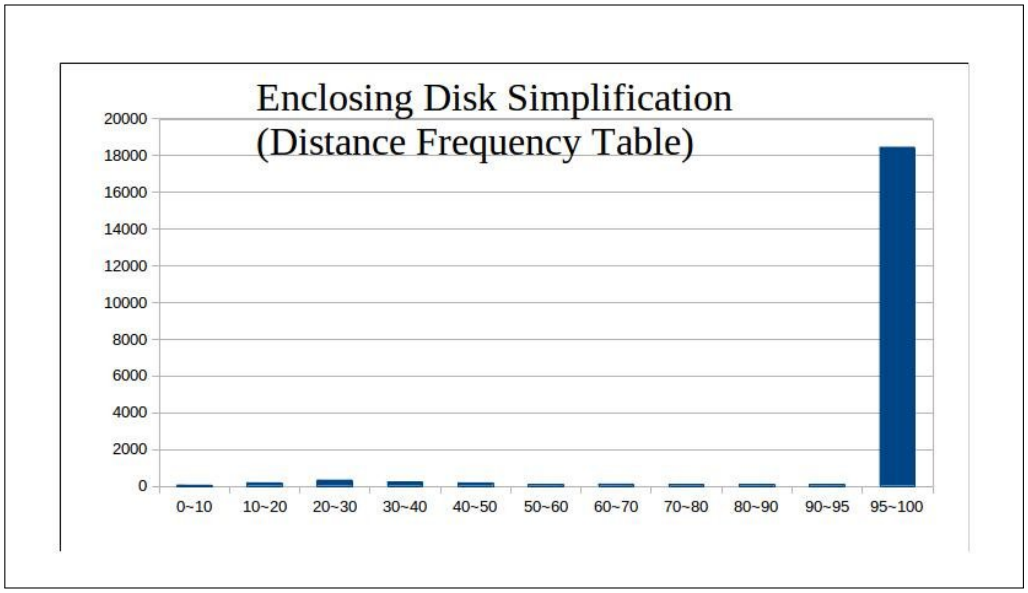

Figure 7.2: This chart shows the error frequency of the entire dataset after applying the enclosing disk simplification method.

We ran both of the trajectory simplification methods for the entire dataset and computed their errors. We used the scaled vectors in our dataset and the $\epsilon = 100$ as our threshold. The entire dataset is a long list and therefore we show the frequency of these errors (Figure 7.2 Figure 7.3).

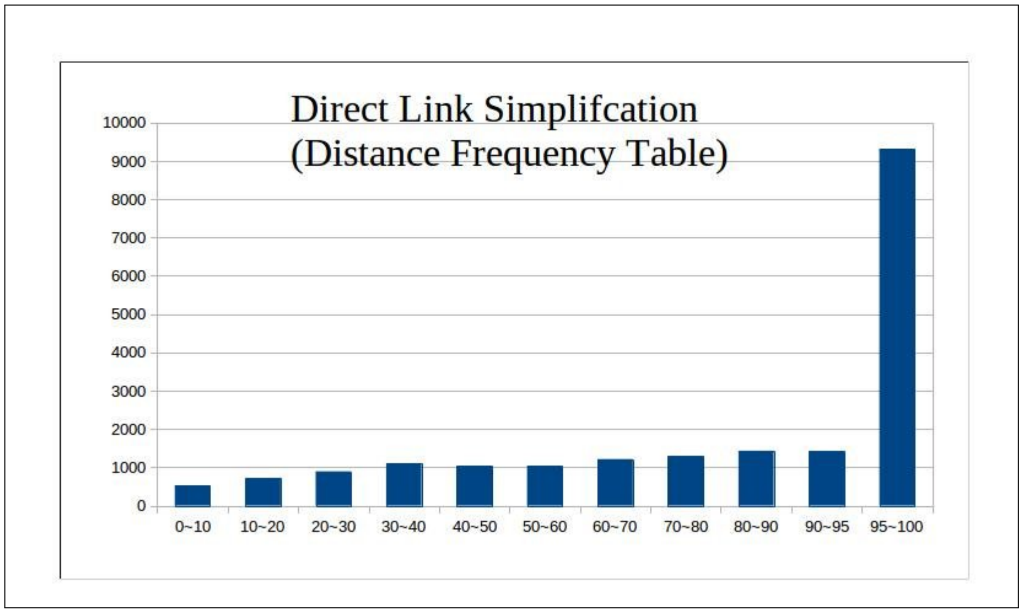

Figure 7.3: This chart show the error frequency of the entire dataset after applying the direct link simplification method.

It is clear that the error caused by enclosing disk method simplification is greater than 95 units for more than 18000 trajectories (refer to Figure 7.2). This means more than 18000 of the trajectories have an error distance larger than $.95\epsilon$. On the other hand, the direct link simplification method caused an error greater than 95 units for more than 9000 trajectories. This is almost half of the dataset population. For the other half of the dataset trajectories, the error caused by direct link simplification is distributed almost uniformly with lower values. This shows that the direct link simplification method performs slightly better than the enclosing disk method for simplification error.

7.3.3 Size-Ratio Distribution

Size-Ratio is the last metric we use to compare the two simplification metric. Size-Ratio is the ratio between the size of the simplified version of a trajectory and the size of the original trajectory. In this context, size is the number of spatio-temporal nodes in a trajectory. Let S_ϵ be a simplification method with a threshold, ϵ, that maps a trajectory T to its simplified version $S_\epsilon(T) = T'$. The Size-Ratio of applying S_ϵ to T is equal to $\frac{|T'|}{|T|}$. Clearly smaller values for size-ratio implies that the simplification algorithm performed better. Recall the main reason we simplified the trajectories was to reduce the number of spatio-nodes describing a trajectory. We ran both the simplification methods and collected data points to compute the size-ratio for each trajectory after applying each simplification method. We will present the size-ratio frequency distribution for both methods here. Before presenting the distribution charts, let us look at some data points that were calculated after applying these simplification methods:

Simplification Method	**Node Count Before Simplification**	**Node Count After Simplification**	**Ratio**
Enclosing Disk	17752226	942898	0.0531
Direct Link	17752226	1567460	0.0883

Table 7.5: The second column is the aggregated number of nodes of all the trajectories before the simplification. The third column is the aggregated number of nodes of all the simplified trajectories. The forth column is the ratio between the third and the second column.

We ran both of the simplification methods and counted to total aggregated number of spatio-temporal nodes before and after the simplification and computed the ratio between them. This reveals that the enclosing disk simplification computes a simplified trajectory of less than 6 percent of the original trajectory size on average. This metric is just below 9 percent for the direct link method. We can see that the smallest enclosing disk method performs slightly better in terms of reducing the number of spatio-temporal nodes of a trajectory. Figure 7.4 shows the size-ratio distribution of all the trajectories after applying both simplification methods. It is worth noting that the threshold, ϵ, used for this experiment was again 100. The smallest enclosing disk method has reduced the number of nodes to less than 1 percent of the original size for 17212 trajectories. That is more than 86 percent of the trajectories have shrunk to less than 1 percent of their original size. On the other hand, the direct link simplification method achieved the same results for 12092 trajectories which is about 60 percent of the trajectories. It is clear that both simplification methods are very effective but enclosing disk method achieves much better results in reducing the number of points from a given trajectory.

7.3.4 Conclusion

It is clear that both trajectory simplification methods are effective and suitable for our task. These results show that there is a trade off between the error factor and the size-ratio factor. Enclosing disk method error rate was higher than the direct link method. On the other hand,

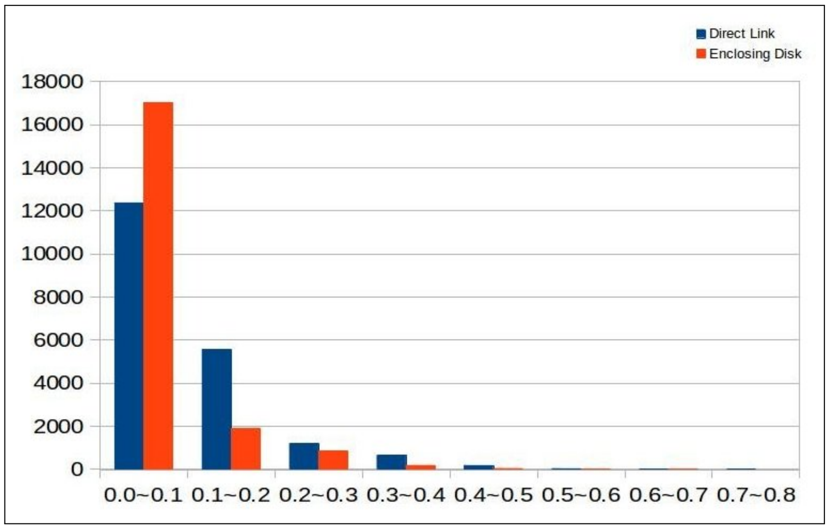

Figure 7.4: The chart above compares the frequency of size-ratio analysis for both of the simplification methods.

the direct link method scored much lower in terms of reducing the number of trajectory nodes. The running time of both of the algorithms is quiet fast and practical. We can conclude that the end-user can decide to go with either of the algorithms based on the precision they would like to achieve from the algorithm. In order to reduce the error factor, it is better to go with the direct link method and choose a smaller threshold. On the other hand, if space and memory resources are limited then the enclosing disk method is more suitable.

Additionally, we performed one more experiment in order to show the effect of simplification on the distance computation time. Recall the experiment in Section 7.2 where we showed the time it took to compute the distance between randomly chosen pairs of trajectories. We run the same experiment this time using the simplified trajectories obtained from direct link method. The results are shown in Table 7.7. Recall that in Section 7.2, we showed that the linear algorithm for computing trajectory distance runs at 0.617 milliseconds on average for a randomly chosen pair of trajectories. After applying the direct link simplification above, this rate drops to 0.05 millisecond *i.e.*, the distance computation is more than 12 times faster than before. This is crucial for clustering algorithms as they frequently make use of the distance function.

We have also included all the simplified trajectories into two images that are presented in Figure 7.5 and Figure 7.6.

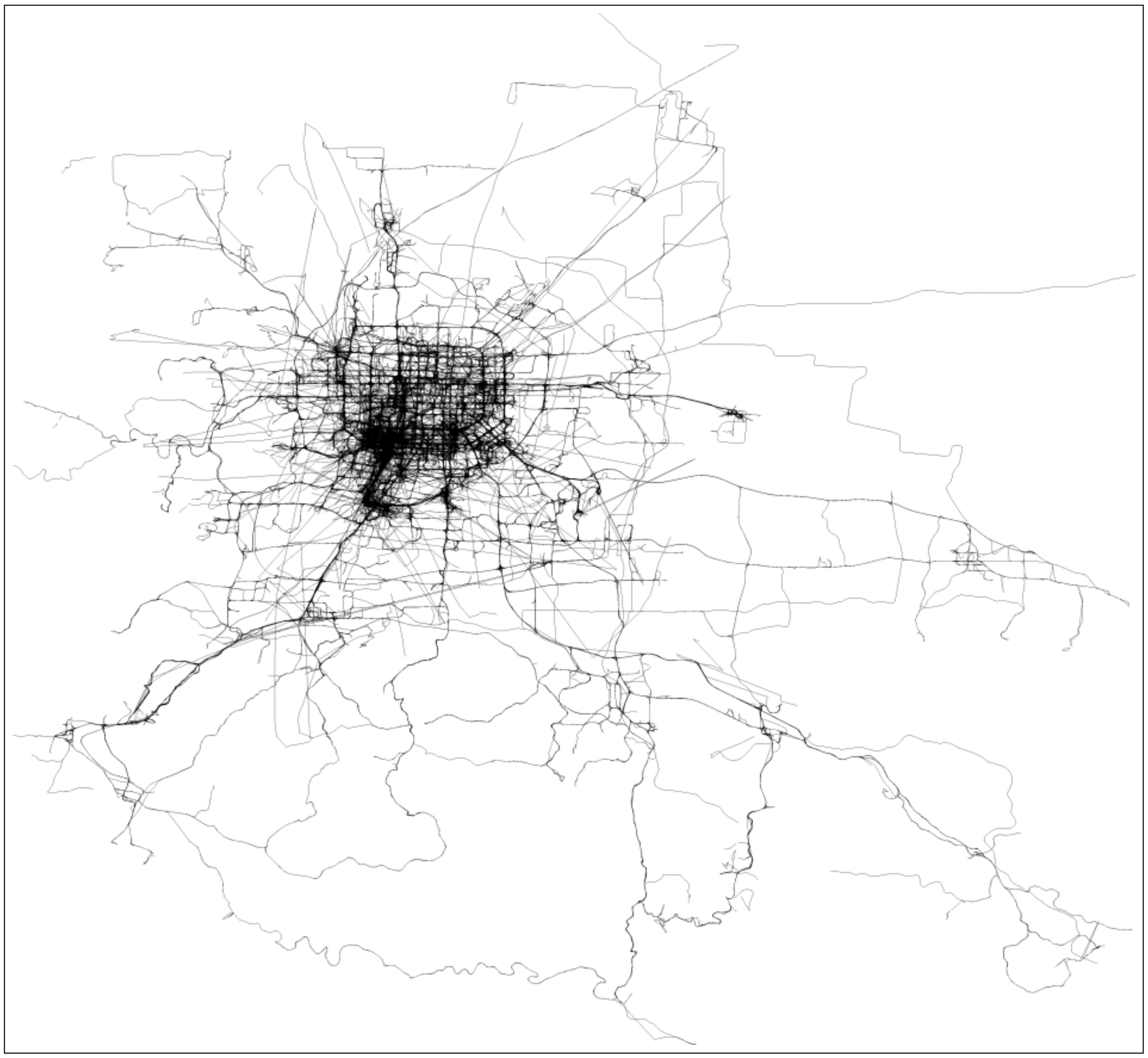

Figure 7.5: Simplified trajectories using enclosing disk method in a single image.

7.4 Clustering Experiments

Finally, we compare different clustering algorithms to cluster the simplified trajectories. In this section, we compare three clustering algorithms: HAC, BUBBLE and BUBBLE-FM. We used the results of direct link simplification algorithm. Here is the running time of these three algorithms:

Notice how much faster BUBBLE and BUBBLE-FM perform in comparison with HAC. These results are expected because HAC runs in quadratic time with respect to the number of items to cluster. On the other hand, BUBBLE and BUBBLE-FM are almost linear-time

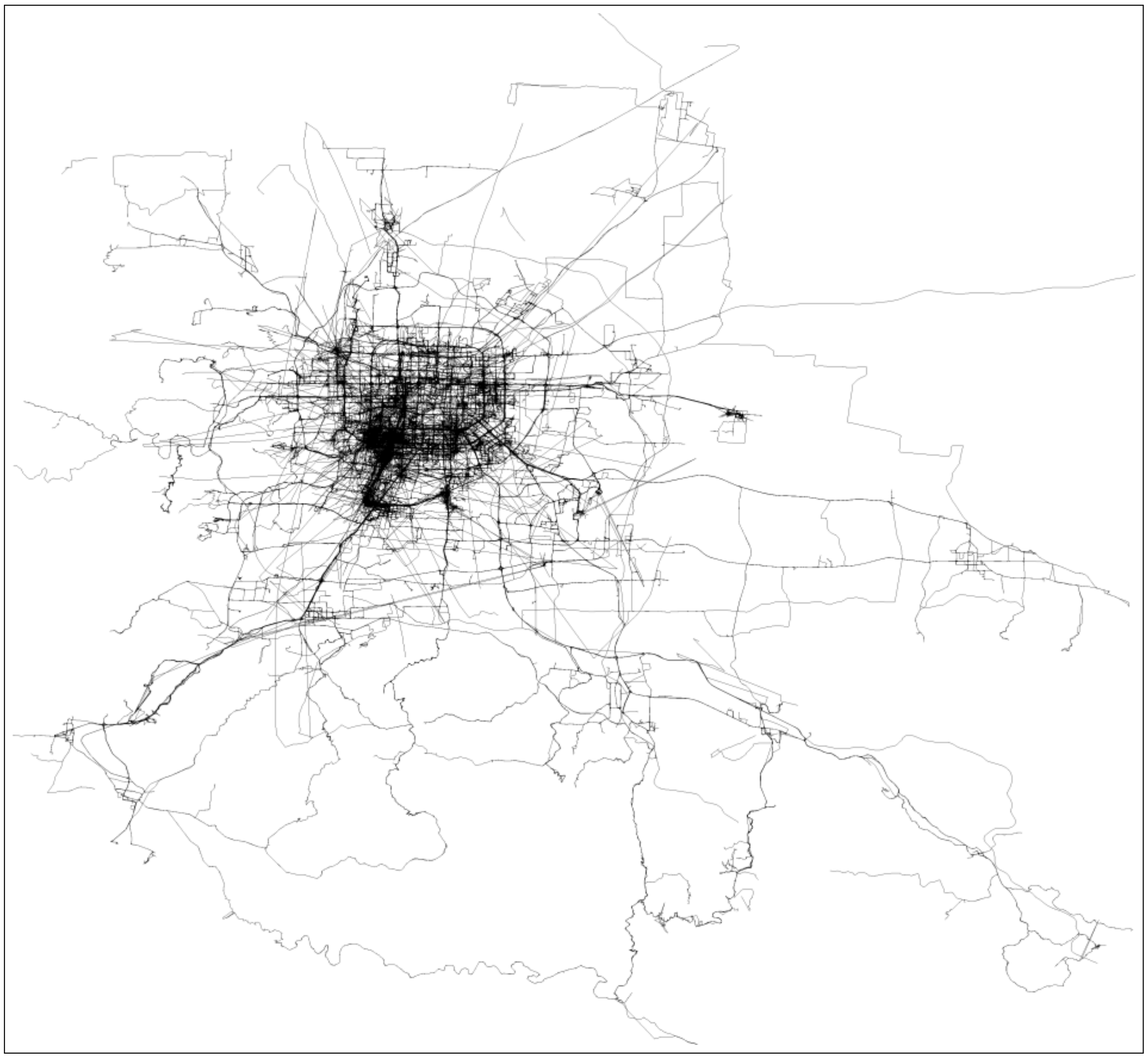

Figure 7.6: Simplified trajectories using direct link method in a single image.

algorithms. We have included snapshots of some of the clusters Figure 7.8. Please note that these images are scaled to fit in the page.

7.5 Conclusion

In this chapter, we compared different algorithms that we introduced in this thesis. It was expected to observe that the running time of the linear distance function to be much better than the quadratic distance function. In this chapter, we put these algorithms to test and showed that in practice the linear algorithm runs much faster than its counterpart. This is crucial for a real

Number of trajectory pairs	**Total running time (Linear Algorithm)**	**Total running time Quadratic Algorithm**
14622	$761ms$	$5,290ms$
7058	$352ms$	$2,451ms$
3856	$193ms$	$1,315ms$
8424	$422ms$	$2,893ms$
12009	$611ms$	$4,400ms$
16382	$836ms$	$5,952ms$
1670	$91ms$	$641ms$
15093	$762ms$	$5,351ms$
11896	$629ms$	$4,388ms$
17969	$907ms$	$6,287ms$

Table 7.6: Results of running the experiment in Section 7.2 with simplified trajectories.

Algorithm	**Running Time (in milliseconds)**	**Running Time (in minutes)**
HAC	12152089	202.53
BUBBLE	339541	5.66
BUBBLE-FM	299595	4.99

Table 7.7: Running times of different clustering algorithms to cluster the entire input of simplified GeoLife dataset.

time clustering algorithm. We also saw that the simplification methods can help reduce the size of trajectories with very little affect. Comparing trajectories can be very time-consuming if a substantial number of comparison has to be made. Reducing the number of nodes makes the clustering algorithm run much faster. We observed the effectiveness of the simplification methods. Finally, we compared the clustering algorithms and observed that the incremental clustering algorithm runs much faster than its counter parts and provides reasonable results.

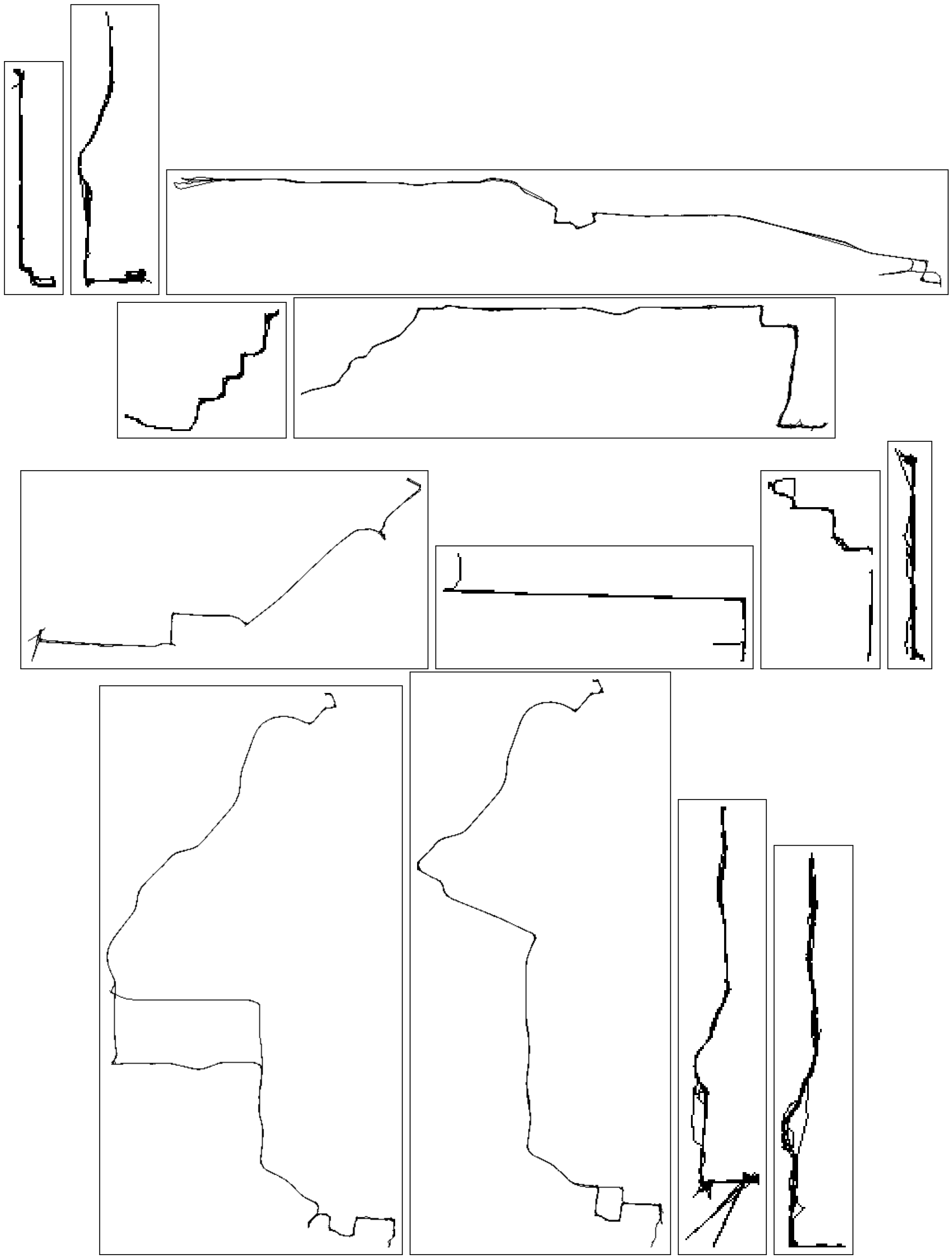
Figure 7.7: Some example of the clusters obtained from BUBBLE-FM.

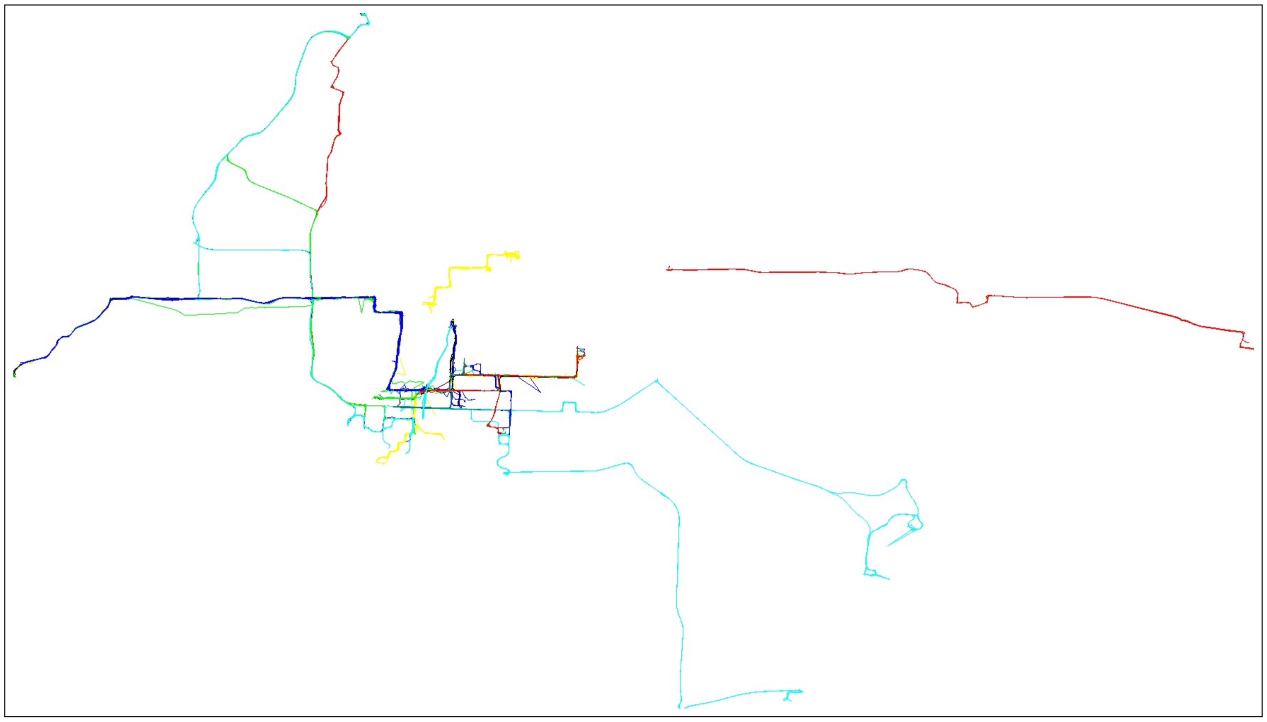

Figure 7.8: Snapshot of some of the clusters obtained from BUBBLE-FM.

Chapter 8

Conclusion

8.1 Summary

In this thesis, we investigated a solution for clustering trajectories in real time. This task is very challenging because of the way data must be presented. We started off by investigating how to compare trajectories. Clustering algorithms require a metric to be defined so that the objects can be compared against each other. We started off by investigating the Fréchet distance function. This metric is widely applied when comparing polygonal chains. On the other hand, this method is not concerned with the speed of trajectories.

We showed an extension of Fréchet distance and we used it to compare trajectories. We illustrated two algorithms to compute this distance accurately. The first algorithm was quadratic and therefore it was not practical. The second method was a linear running time algorithm which was suitable for computing the distance between trajectories in practice.

Furthermore, we introduced the problem of long trajectories. We presented two methods to simplify trajectories such that they would contain less nodes. This makes the distance function run much faster. Additionally, the simplified trajectories are easier to store in memory or a storage device.

We investigated a few clustering algorithms including HAC. Most of the well-known clustering algorithms such as k-means work with feature vectors. These clustering algorithms are also limited because the entire sample set must be provided to them prior to their execution. This is clearly not the case with a real-time system. On the other hand, we did not have feature vectors but we defined our own distance function. Thus, we needed an incremental clustering algorithm that would work with a customized distance function. We took advantage of the fact that our distance function was a metric function. We showed that we can cluster trajectories in real-time using BUBBLE and BUBBLE-FM algorithms.

8.2 Future Work

This thesis presented a valid way of clustering trajectories based on their position and velocity. The distance function introduced here does not account for the size of the object or the colour histogram of the moving object. These measures can be useful when the clustering algorithm is targeted for surveillance applications. This work can be extended to account for such parameters.

Additionally, we presented a 2D clustering solution. However, the same approach can be taken to cluster trajectories that are defined using 3D spatio-temporal nodes. Currently, the smallest enclosing disk simplification method used in this work cannot be applied in a 3D setting. In that case, a 3D sphere must be computed. However, the direct-link method can be applied to simplify 3D trajectories. We had no access to a 3D trajectory dataset to perform experiments to see if these methods can still be reliable. It would be interesting if these methods could be applied in a 3D setting and see if they can be optimized for higher dimensions.

8.3 Conclusion

This thesis shows an efficient way for clustering trajectories in real time systems. It shows how we can simplify and store trajectories in a database while they are being discovered. For this to be possible in real time, the trajectories are required to be passed through a simplification process. This makes it possible for us to keep as many trajectories in memory. More importantly, it makes the distance computation between trajectories a very efficient process that can be applied in a real world problem. Moreover, in order to group trajectories together, one needs a metric to compare trajectories. We defined a derivation of Fréchet polygonal chain distance and applied it to identify a new metric for comparing trajectories.

Real time systems are always limited in resources and therefore a clustering algorithm was used to both maintain clusters in memory and an external storage place such as a database. Clustering trajectories in surveillance systems can be crucial. Identifying trajectories that fit in the same category can help a system learn about the trajectory patterns. Surveillance systems can exploit these patterns to learn how objects commonly move through space. This can be crucial to detect the suspicious objects moving through space.

Throughout this work, we were faced with so many different challenges and limitations of a real-time system. We overcame these challenges by designing very efficient algorithms to simplify and compute distances between trajectories. This is a practical solution and it can be deployed in a real-world surveillance system.

References

[1] Helmut Alt and Michael Godau. Measuring the resemblance of polygonal curves. In *Proceedings of the eighth annual symposium on Computational geometry*, SCG '92, pages 102–109, New York, NY, USA, 1992. ACM.

[2] Sergey Bereg, Minghui Jiang, Wencheng Wang, Boting Yang, and Binhai Zhu. Simplifying 3d polygonal chains under the discrete fréchet distance. In *Proceedings of the 8th Latin American conference on Theoretical informatics*, LATIN'08, pages 630–641, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] John Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, June 1986.

[4] Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16:361–368, 1996.

[5] W. S. Chan and F. Chin. Approximation of polygonal curves with minimum number of line segments. In *Proceedings of the Third International Symposium on Algorithms and Computation*, ISAAC '92, pages 378–387, London, UK, UK, 1992. Springer-Verlag.

[6] P. Cheeseman, M. Self, J. Kelly, J. Stutz, W. Taylor, and D. Freeman. AutoClass: a Bayesian classification system. In *Machine Learning: Proceedings of the Fifth International Workshop*. Morgan Kaufmann, 1988.

[7] Alon Efrat, Leonidas J. Guibas, Sariel Har-Peled, Joseph S. B. Mitchell, and T. M. Murali. New similarity measures between polylines with applications to morphing and polygon sweeping. *Discrete & Computational Geometry*, 28(4):535–569, 2002.

[8] Thomas Eiter and Heikki Mannila. Computing discrete fréchet distance. Technical report, Christian Doppler Laboratory for Expert Systems,, TU Vienna, Austria, 1994. Technical Report CD-TR 94/64.

[9] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Evangelos Simoudis, Jiawei Han, and Usama M. Fayyad, editors, *KDD*, pages 226–231. AAAI Press, 1996.

[10] Christos Faloutsos and King-Ip Lin. Fastmap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. *SIGMOD Rec.*, 24(2):163–174, May 1995.

[11] M. Fréchet. *Sur quelques points du calcul fonctionnel, par M. Maurice Fréchet,...* Palermo (30 via Ruggiero Settimo), 1906.

[12] Venkatesh Ganti, Raghu Ramakrishnan, Johannes Gehrke, and Allison Powell. Clustering large datasets in arbitrary metric spaces. In *Proceedings of the 15th International Conference on Data Engineering*, ICDE '99, pages 502–, Washington, DC, USA, 1999. IEEE Computer Society.

[13] Trevor Hastie, Robert Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. New York: Springer-Verlag, second edition, 2013.

[14] F. Hausdorff. *Grundzüge der mengenlehre*. Veit & co., 1914.

[15] Hiroshi Imai and Masao Iri. Computational-geometric methods for polygonal approximations of a curve. *Comput. Vision Graph. Image Process.*, 36(1):31–41, November 1986.

[16] Minghui Jiang, Ying Xu, and Binhai Zhu. Protein structure-structure alignment with discrete fréchet distance. *J. Bioinformatics and Computational Biology*, 6(1):51–64, 2008.

[17] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[18] C. Kenyon-Mathieu and V. King. Verifying partial orders. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, STOC '89, pages 367–374, New York, NY, USA, 1989. ACM.

[19] Lawrence A. Klein. *Traffic Detector Handbook*. Federal Highway Administration, third edition, 2006.

[20] Joseph B. Kruskal and Myron Wish. *Multidimensional scaling*. Sage university papers, Quantitative applications in the social sciences; 11. Sage Publ., Newbury Park, Calif. [u.a.], [nachdr.] edition, 1994.

[21] Ralph Lange, Frank Dürr, and Kurt Rothermel. Online trajectory data reduction using connection-preserving dead reckoning. In *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, Mobiquitous '08, pages 52:1–52:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[22] Ralph Lange, Tobias Farrell, Frank Durr, and Kurt Rothermel. Remote real-time trajectory simplification. In *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications*, PERCOM '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[23] J. MacQueen. Some methods for classification and analysis of multivariate observations. Proc. 5th Berkeley Symp. Math. Stat. Probab., Univ. Calif. 1965/66, 1, 281-297 (1967)., 1967.

[24] Nimrod Megiddo. Linear-time algorithms for linear programming in r3 and related problems. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:329–338, 1982.

[25] J. O'Rourke. In *Polygonal chain approximation: An improvement to an algorithm of Imai and Iri*. Department of Electrical Engineering and Computer Science, John Hopkins University, 1985.

[26] Jonathan Owens, Andrew Hunter, and Eric Fletcher. A fast model-free morphology-based object tracking algorithm. In *BMVC*, pages 1–10, 2002.

[27] G. Papakonstantinou. Optimal polygonal approximation of digital curves. *Signal Processing*, 8(1):131–135, February 1985.

[28] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, SFCS '75, pages 151–162, Washington, DC, USA, 1975. IEEE Computer Society.

[29] E. Sriraghavendra, Karthik K., and C. Bhattacharyya. Fréchet distance based approach for searching online handwritten documents. In *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 01*, ICDAR '07, pages 461–465, Washington, DC, USA, 2007. IEEE Computer Society.

[30] J. J. Sylvester. A question in the geometry of situation. *Quarterly Journal of Pure and Applied Mathematics*, 1:79, 1857.

[31] Warren Torgerson. Multidimensional scaling: I. theory and method. *Psychometrika*, 17(4):401–419, December 1952.

[32] G. Trajcevski, H. Cao, P. Scheuermann, O. Wolfson, and D. Vaccaro. Online data reduction and the quality of history in moving objects databases. In *Processding of the 5th ACM Internation Workshop on Data Engineering for Wireless and Mobile Access*. ACM, 2006.

[33] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In *Results and New Trends in Computer Science*, pages 359–370. Springer-Verlag, 1991.

[34] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 103–114. ACM Press, 1996.

[35] Yu Zheng, Quannan Li, Yukun Chen, Xing Xie, and Wei-Ying Ma. Understanding mobility based on gps data. In *Proceedings of the 10th international conference on Ubiquitous computing*, UbiComp '08, pages 312–321, New York, NY, USA, 2008. ACM.

[36] Yu Zheng, Lizhu Zhang, Xing Xie, and Wei-Ying Ma. Mining interesting locations and travel sequences from gps trajectories. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 791–800, New York, NY, USA, 2009. ACM.