Algorithmic and Experimental Results on Trajectory Data Processing

The cover image was generated by plotting trajectory data from OpenStreetMap.

This thesis has been typeset using the LaTeX typesetting system with the fonts Linux Biolinum and Linux Libertine.

Printing: Ridderprint, https://www.ridderprint.nl

ISBN: 978-90-393-7450-4

Algorithmic and Experimental Results on Trajectory Data Processing

Algoritmische en Experimentele Resultaten over het Verwerken van Trajectory-data
(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit Utrecht
op gezag van de
rector magnificus, prof.dr. H.R.B.M. Kummeling,
ingevolge het besluit van het college voor promoties
in het openbaar te verdedigen op

woensdag 30 maart 2022 des middags te 2.15 uur

door

Mees Anton van de Kerkhof

geboren op 29 juli 1993
te Beuningen

Promotor:

Prof. dr. M.J. van Kreveld

Copromotoren:

Dr. M. Löffler
Dr. I. Kostitsyna

Dit proefschrift werd (mede) mogelijk gemaakt met financiële steun van HERE Technologies en de NWO onder projectnummer 628.011.005

Contents

Chapter 1

Introduction

In the years since the launch of the Global Positioning System (GPS) and other Global Navigation Satellite Systems (GNSS), and their subsequent opening up to non-military use, satellite navigation technology has proven to be incredibly influential. It has transformed the way we perform everyday tasks such as ordering food or navigating our cars. GPS technology is everywhere. Many so-called *location-based services*, such as Uber, Tinder, Pokémon Go, etc., have been successfully launched using satellite navigation technology and many more possible services and applications are on the horizon. Besides the location-based services, which are mostly online, there is also an increasing amount of GPS data being collected for offline analysis. GPS trackers are used to collect movement data of animals such as pigeons, elk, and deer. Commercial ships have their movements logged by GPS to collect data usable for ocean traffic planning, and users of services like Google Maps can opt to share their data so it can be analyzed to improve the map and the navigation software.

This proliferation of GPS use has resulted in previously unprecedented amounts of GPS data being generated. In turn, this has caused great interest in algorithms for preprocessing and analyzing all of this data. Such algorithms are the subject of this thesis.

Besides GPS itself, other navigation satellite networks have also been launched, such as the European Union's Galileo, or Russia's GLONASS. Other ways of tracking movement, such as Radio-Frequency Identification (RFID) or Wi-Fi tracking are now also on the rise [75]. These do not work in the same way as GPS, being based on local scanning infrastructure rather than far-away satellites, but they generate data

that can be analyzed similarly to GPS data. In this thesis, we will focus on navigation satellite data for tracking movement. We use the terms GPS, GNSS, and navigation satellites interchangeably.

GPS data is generated by attaching a GPS tracker to some entity whose movement we are interested in. At certain times, the GPS tracker will deduce its position by measuring the distances between itself and the GPS satellites it has a clear line of sight to. The atomic unit of GPS data is the *probe*. This is one measured location for one entity at one point in time. Depending on the setup, additional data, such as the speed or heading of the entity at the time of measurement, may also be recorded. Multiple probes tracking the movement of the same entity are collected in a *trajectory*. A trajectory consists of a set of probes tracking the same entity, sorted chronologically. For times in between probes, where there is no measured position available, the position can be estimated by interpolation. There has been some research done into different interpolation schemes [102], but in general, if only having data at the measured times is insufficient, linear interpolation between the probes is used. Likewise, if the heading or speed were not measured for each probe, they can be estimated by assuming a constant speed and direction in between probes. This does mean that the speed and heading do not change continuously but change abruptly at discrete times. For most trajectory applications, this is an acceptable approximation. When using linear interpolation like this, we can envision a trajectory as a polygonal curve, where each vertex is annotated with a timestamp and additional data depending on the use case. See Figure 1.1. For some use cases, the timestamps may be ignored. The trajectories are then treated purely as polygonal curves without annotations. Although this causes some loss of information, for some use cases this does not matter. For example, for map construction the shape and location of the trajectories is important, but the time the trajectory was captured or the speed of the entity do not matter as much. Ignoring the timestamps may allow for more straightforward algorithm design, and it also opens up the use of existing research on algorithms that operate on polygonal curves. For example, curve simplification methods can be used to simplify trajectories.

Nearly all GPS algorithms focus on trajectories, rather than individual probes. The input for these algorithms is thus a *trajectory data set.*

1.1 Trajectory tasks

The computational geometry and GIS communities have formulated many interesting questions and problems involving trajectories. To better understand how these problems and the algorithms that solve them interplay we can loosely categorize the

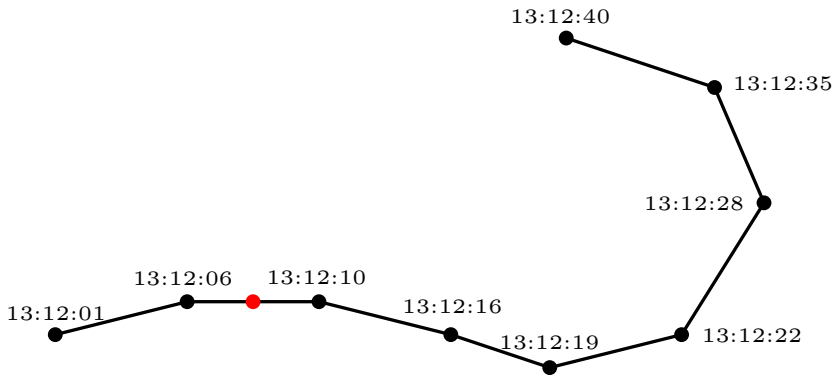


Figure 1.1: A set of probes (positions annotated by a timestamp) together form a trajectory. For times where no probe exists we can find one by interpolation, like the virtual probe at time 13:12:08 shown in red.

problems into *preprocessing tasks*, *tools* and *application tasks*. Preprocessing tasks are tasks that can be applied to many different trajectory data sets as a precursor to being analysed, to remove common errors and reduce storage and subsequent computation times. Tools are more specific, being used only when the use case specifically calls for it, but in most cases they are not the last task to be performed on the data. They can also form a subroutine for a larger task. Application tasks are where the already preprocessed data is used to extract useful information or perform some task for the end user, such as finding movement patterns or constructing a road network. It is helpful to split up tasks like this, rather than designing application algorithms to handle unpreprocessed data. By splitting the tasks up, a preprocessing algorithm can be reused for different use cases. The data can be stored in a preprocessed manner, which may require less storage, and the application algorithms can make stronger assumptions on the input, giving more options for designing algorithms. A non-exhaustive overview of trajectory tasks is given below. A categorization of the tasks is given in Table 1.1.

Trajectory distance / similarity One elementary task that comes up during several other trajectory tasks, is being able to compute a measure of distance between

Table 1.1: Categorization of trajectory tasks.

Preprocessing tasks	Tools	Application tasks
Simplification	Similarity	Finding regions of interest
Outlier detection	Segmentation	Map construction
Stay point detection	Clustering	Map generalization
Anonymization	Map matching	

two trajectories. Here we mean distance as in how much one trajectory must be changed before it is identical to another trajectory, i.e. the inverse of the similarity between trajectories. The high dimensionality of trajectory data makes this less straightforward than finding the distance between simpler objects, such as two points. Well-known measures are the Hausdorff distance [89], Fréchet distance [16], and Dynamic Time Warping [55]. New measures and variations on existing measures are still regularly introduced, such as versions of the Fréchet distance that take context into account [38], versions that allow the trajectory to be split up and recombined before computing the distance [13], or versions that consider the uncertainty inherent to trajectory data [35]. Most of the proposed distance measures do not take the time information of trajectories into account, only considering the shape of the trajectory. This also allows the measures to be used more generally as distance measures for polygonal curves, rather than specifically focusing on trajectories. There do exist distance measures specifically for trajectories, including the time stamps. One example is the Time-Synchronized Euclidean distance [106]. A recent comparison between several trajectory distance measures was made in [124].

Trajectory simplification Some trajectory data sets contain more probes than are needed to capture the movement information, especially since probes can be approximated by linear interpolation of the adjacent probes. Trajectory simplification is the reduction of the number of probes in a trajectory to reduce memory requirements and speed up further algorithms. It is important that the simplified trajectory still resembles the original. For this, trajectory distance measures can be used. The goal is then to reduce the number of probes as much as possible while having a bound on the curve distance between the original and the simplification. This is known

as min-link simplification. Alternatively, we could set a bound on the number of probes of the simplified trajectory and try to find a simplification with minimum distance, known as a min-δ-simplification. Of course, one can also simplify without necessarily optimizing one or the other. The widely used Douglas-Peucker [49] algorithm, for example, guarantees the simplification is within a pre-specified distance δ, but provides no guarantee of optimality in probe-reduction. A comparison of Douglas-Peucker and other simplification algorithms with no optimality guarantee is given in [105]. There are many different algorithms for different approaches, such as online simplification [98], different restrictions on where the simplification's probes can be placed, etc. [6, 9, 21, 25, 36, 39, 42, 63, 72].

Outlier detection One issue arising when working with trajectories is the fallibility of GPS. In order to get a good measurement of the location, the GPS needs a clear line of sight to as many satellites as possible, requiring at least four. Lack of satellites, or miscalculation of the distance to a satellite as a result of the signal being reflected or distorted means that there is always a degree of uncertainty in the location of the entity. Even in prime conditions we can still expect a deviation of a few meters to the actual location when using a standard GPS receiver, with larger errors occasionally also occurring [1]. A small error is generally acceptable (and unavoidable), but these larger errors can cause bigger problems as they are likely to distort the result. For example, one single outlier can dramatically increase the computed value for the Fréchet distance. Therefore it is important to deal with these measurements. Outlier detection is of course an issue when dealing with a data set of any type, so general methods exist [69, 68], but a significant amount of research has also been done in considering the case for trajectories specifically. One possibility is to apply smoothing to the trajectory, where the location measurements for the probes are shifted to bring them closer together using techniques such as a Kalman filter or a particle filter [94]. Alternatively, the outlying probes can be detected and removed from the trajectory either heuristically [139], or by comparison to other data. For example, we can look at multiple similar trajectories at once to identify probes lying far away from the other trajectories [92]. Many methods consider the whole trajectory to be the "outlier" in this case, rather than a single probe [62, 92, 96, 133]. If a road network is available, this can help identify outlying probes as they may lie outside of the road network or have a large graph distance to adjacent probes. In these cases, outlier detection does not have to be a separate step from map-matching if the errors are small enough. In the cases where single probes are seen as outliers rather than entire trajectories, detecting and removing outliers has some resemblance to trajectory simplification. However, simplification

generally aims to minimize the number of probes while still accurately describing the trajectory: this typically retains outliers as these are "salient".

Trajectory segmentation Some trajectory data sets contain large trajectories that can be better analyzed by dividing them into smaller pieces to be studied separately. Trajectory segmentation is the process of splitting up a trajectory into contiguous disjoint segments based on some criterion. For example, we may want the probes in each segment to have a speed that is at most twice as large as any other probe in the segment [19]. Or if the trajectory is of a person travelling using different modes of transport, we may want to segment the trajectory so each leg of the trip gets its own segment [141].

Trajectory clustering While trajectories in the same data set generally share some characteristics, such as being collected during the same timespan, or depicting vehicles of the same type, they can still be very diverse in physical location and shape. If the task we want to perform on the trajectories requires the trajectories to be similar, we may want to divide the data set into clusters, where trajectories in the same cluster are closer to each other in space and time than they are to trajectories not in the cluster. Data clustering is used in many domains, but algorithms used in other domains mainly operate on point data, making it so they cannot be straightforwardly applied to trajectories [93]. Some trajectory clustering methods assign the full trajectories to clusters [117, 56], but we may also want to cluster subtrajectories [30, 12, 93] instead, as longer trajectories might be dissimilar even if they have many subsections that are similar. Once the trajectories are clustered, a *representative trajectory* can be chosen or constructed to represent the general shape of the trajectories in the cluster [33].

Stay point detection / finding regions of interest For many entities it is natural that at some time, they will rest and stop moving. Elk and deer we might be tracking will go to sleep, cars we are tracking are parked and not used for some time. If the GPS tracker keeps generating new probes during this time, due to the error inherent in GPS technology, they may not all be measuring the same location even as the entity remains motionless. These *stay points* can cause trouble for follow-up algorithms. So some algorithms focus on detecting these stay points so that they can be removed. Stay points can also be detected as a method for finding *regions of interest* to be added to an annotated map, such as popular tourist spots [128]. Since the movement characteristics of the entity are different during a stay point as compared to when the entity is going somewhere, they can be found using trajectory

segmentation methods, although there also exist algorithms specifically for finding them [95, 66, 65].

Map matching If we are given a set of trajectories as well as an embedded graph of the underlying road network, we can match the trajectories to the road network . This means we edit the trajectories' probes such that the trajectory now lies on the road network [15, 112]. This can be done to bring the trajectories closer to expected reality, as we expect cars to keep on the road. It can also be done to convert the trajectory to a path in the graph, for example, to count how many trajectories cover a certain road segment (graph edge). For an overview of map-matching techniques, see [139].

Anonymization Digital privacy has been a topic of increasing interest. To protect privacy, trajectory information cannot always be published as-is. It is important that when published trajectory data was generated by tracking people, it is not possible for attackers to deduce which person was tracked for any specific trajectory. This is a difficult task, as removing or modifying data that could be identifying also has an impact on the usability of the data set, so a good trade-off must be made. One step that is often done in data sets is removal of some of the probes at the start and end of the trajectory. These are likely to contain the departing point and destination of the tracked person, and so this identifying information is removed. Further anonymization can be done by applying the principle of *k-anonymity*, i.e. generalizing and removing trajectories such that each remaining trajectory is identical to at least $k - 1$ other trajectories [120]. Machine learning approaches to trajectory privacy have also been proposed [100]. Stay point removal or generalization of stay points [71] can also be an important step in preserving privacy when publishing trajectory data, as stay points may help malicious actors deduce where important locations are. For example, the location of a warehouse could be discovered by looking for the stay points of delivery trucks.

Map construction Internally, most applications that deal with a road network store it as a graph, but it is difficult to figure out what this graph should be and to also keep it up to date. One possible application task for a trajectory data set is thus constructing a graph of the underlying road network, if no previous graph is available, or to identify changes to the road network and update the graph if one is available [31]. A comparison between different map construction algorithms utilizing GPS data can be found in [10].

Map generalization Besides serving to construct a graph, we can also use trajectories as a means of annotating a graph with specific data. Encoding the traffic flow into weights on the edges and vertices of the graph. This can then be used to generalize the graph, by taking only the subgraph annotated with the most important data. Previous approaches have relied on graph properties [126, 43, 143], but GPS data is now widely available it can be used as well [132].

▸ 1.2 Definitions and notation

A trajectory can be seen as a sorted list of probes, where a probe is a tuple of a location in $\mathbb{R}^2$ and a timestamp $t_i \in \mathbb{R}$:

$$T = \langle (x_1, y_1, t_1), (x_2, y_2, t_2), \cdots, (x_n, y_n, t_n) \rangle \ (t_i < t_{i+1}, i \in \{1, \cdots, n-1\}) \quad (1.1)$$

We can denote the first probe (x_1, y_1, t_1) as p_1, the second as p_2, up till the last probe p_n. However, in many cases we are also interested in the position and speed of the object in between the measured probes. We can get a workable approximation by using linear interpolation between adjacent probes. We can define a function that returns a probe for the trajectory for any real value in the interval $[1, n]$. Given a real value r we split it up into an integer part i and a fractional part λ. This allows us to use the following function to either return one of the given probes, or construct an interpolated probe:

$$T(i + \lambda) = (1 - \lambda)p_i + \lambda p_{i+1} \quad (1.2)$$

We can also envision the set of all constructed probes in between two probes that were part of the input as forming one line segment connecting the two input probes, turning the entire trajectory into a polygonal curve, see Figure 1.1. A *subtrajectory* of a trajectory T, also referred to as a *subsequence*, is an ordered subset of T's given probes, with the subtrajectory's probes having the same relative order as in T. The probes do not need to have been consecutive in T. Algorithmically, a subtrajectory can be treated as a trajectory in its own right.

Below we will give more definitions that are important to know.

▸ 1.2.1 Hausdorff distance

One simple similarity measure for trajectories is the *Hausdorff distance*. It is a very general similarity measure that can be used for any two point sets. If we have two sets of points P and Q, such as two trajectories if we treat them as polygonal curves and disregard the timestamps, the *directed* Hausdorff distance from P to Q is equal

to the Euclidean distance between the point in P that is furthest from any point in Q, and the point in Q that is closest to that point. Written formally, we get:

$$\overrightarrow{\mathsf{H}}(P, Q) = \max_{p \in P} \min_{q \in Q} \|p - q\|$$

The *undirected* Hausdorff distance, also just called the Hausdorff distance, is then the maximum of the directed distances in both directions.

$$\mathsf{H}(P, Q) = \max\{\overrightarrow{\mathsf{H}}(P, Q), \overrightarrow{\mathsf{H}}(Q, P)\}$$

▸ 1.2.2 Fréchet distance and the free space diagram

Another trajectory similarity measure that we use often is the Fréchet distance. It is based on the principle that similar polygonal curves should not just be close in space, but there should also exist some parametrization of the curves such that if we traverse both simultaneously we should remain close at all times. Closeness here is defined as having small Euclidean distance. The trajectories are treated as polygonal curves and the timestamps are not taken into account. For polygonal curves/trajectories P and Q, with n and m probes respectively, the Fréchet distance is defined as:

$$\mathsf{F}(P, Q) = \inf_{(\sigma,\theta)} \max_{j:[0,1]} \|P(\sigma(j)) - Q(\theta(j))\| \tag{1.3}$$

where σ and θ are continuous non-decreasing functions from $[0, 1]$ to the real intervals $[1, n]$ and $[1, m]$, respectively.

This is often explained with the following analogy: Suppose someone is walking their dog. P is the path the owner takes, and Q is the path of the dog. The owner and dog can change their speed at will, but they cannot go backwards on the path. The Fréchet distance is then the shortest possible length the dog's leash can have for this walk to be possible.

The *weak Fréchet distance* is similar, but the constraint is dropped that σ and θ are non-decreasing. Going back to the man-walking-dog analogy this means the man and dog can freely move backwards and forwards over their path if this results in a shorter leash being needed. The weak Fréchet distance between curves thus gives a lower bound for the (strong) Fréchet distance.

The *discrete Fréchet distance* is a variant where σ and θ are discrete functions from $\{0, \ldots, k\}$ to $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$ with the property that $0 \leq \sigma(i+1) - \sigma(i) \leq 1$ and $0 \leq \theta(i + 1) - \theta(i) \leq 1$. This has been explained as someone walking a pet frog,

where instead of walking along edges of the polygonal curve, the owner and frog can only hop from vertex to vertex. The discrete Fréchet distance between two curves gives an upper bound on their (continuous) Fréchet distance.

As you might expect, computing the exact Fréchet distance between two curves is not completely straightforward. Alt and Godau [16] described an approach for computing this distance. They define an algorithm for solving a decision variant of the problem of computing the distance. This algorithm can answer if the Fréchet distance between two curves is at most some value δ. Then they use a technique called parametric search to find the minimum value of δ.

Their decision algorithm works by constructing what is called a *free space diagram* for a value for δ.

It consists of a grid of $(n-1) \times (m-1)$ cells, where each cell corresponds to a pair of line segments, one from P and one from Q. Each column of cells corresponds to an edge of P and each row corresponds to an edge of Q. For example, the point $(2.6, 3.5)$ in a free space diagram corresponds to the probe gotten by linearly interpolating between P's second and third probes with a λ of 0.6, and the probe gotten by linearly interpolating between Q's third and fourth probes with a λ of 0.5. Each cell is divided into *free space* and *forbidden space*. If the Euclidean distance between the curves at a point is less than or equal to δ, the point is in the free space. If the distance is greater than δ, the point lies in the forbidden space. See Figure 1.2.

Now, an x- and y-monotone path from the point $(1, 1)$ to (n, m) entirely through the free space corresponds to parametrizations of P and Q such that the distance between the two is at most δ at any time, i.e. the Fréchet distance is at most δ.

For additional details we refer to the paper by Alt and Godau [16].

▸ 1.3 Contribution

In this thesis, we study a variety of problems that have trajectory data as input. Two of the chapters cover problems relating to preprocessing tasks. We cover outlier detection, giving new algorithms that can be used to remove outliers from trajectories without the need for additional data such as a road network or other trajectories. We study trajectory simplification, going over many variants of this problem and giving either polynomial time algorithms or NP-hardness proofs. One chapter considers a tool: We examine the problem of computing representative trajectories, creating the first known implementation for the algorithm introduced in [90] and testing it on real data. Finally, there is one chapter on an application task. We introduce a novel approach for road network generalization and compare it to existing approaches.

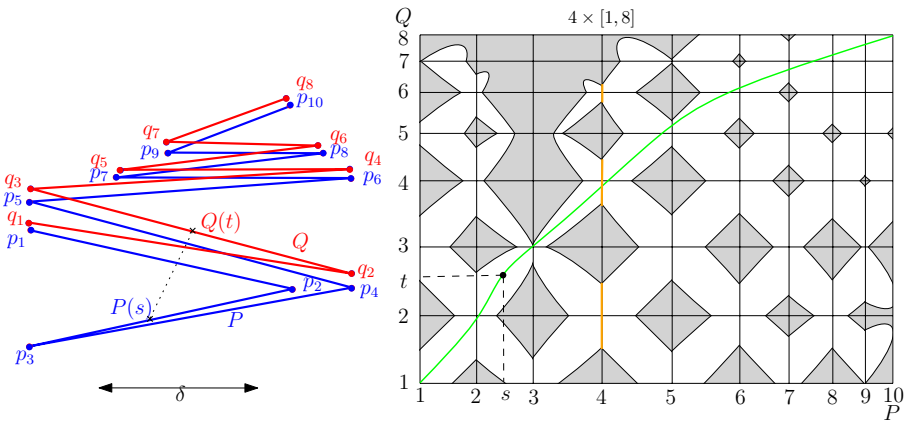


Figure 1.2: Two polygonal curves P (in blue) and Q (in red), and their free space diagram for the chosen value of δ. The free space is shown in white and the forbidden space is shown in gray. Each cell of the FSD corresponds to the combination of one edge of P and one edge of Q. (s, t) is a free point in the diagram, lying on a reachable path in the free space. One green spot is marked in both the FSD and on the associated spots on P and Q. A x- and y- monotone path contained in the free space from (1, 1) to (8, 10), shown in green, corresponds to parametrizations of P and Q realizing a Fréchet distance of at most δ.

1.4 Structure of this thesis

Now we will discuss how the rest of this thesis is set up. After this introductory chapter, there are four chapters, each covering a different trajectory task. After that is a concluding chapter. There are also appendices going over the different data sets that were used and how they were preprocessed. We will now give summaries of the remaining chapters.

Chapter 2: Outlier Detection One of the most fundamental preprocessing steps used for processing trajectories of any type is the detection and correction/removal of outlying probes. Normally, a GPS probe can be expected to lie within a few meters of the actual location of the tracked entity at the time the probe was taken, but a variety of circumstances can create probes that lie much further away. For example, many inner-city environments form what are known as *urban canyons*, where high-rise buildings block the line of sight to the GPS satellites, making it difficult to accurately compute the position of the entity. Outliers can have strong

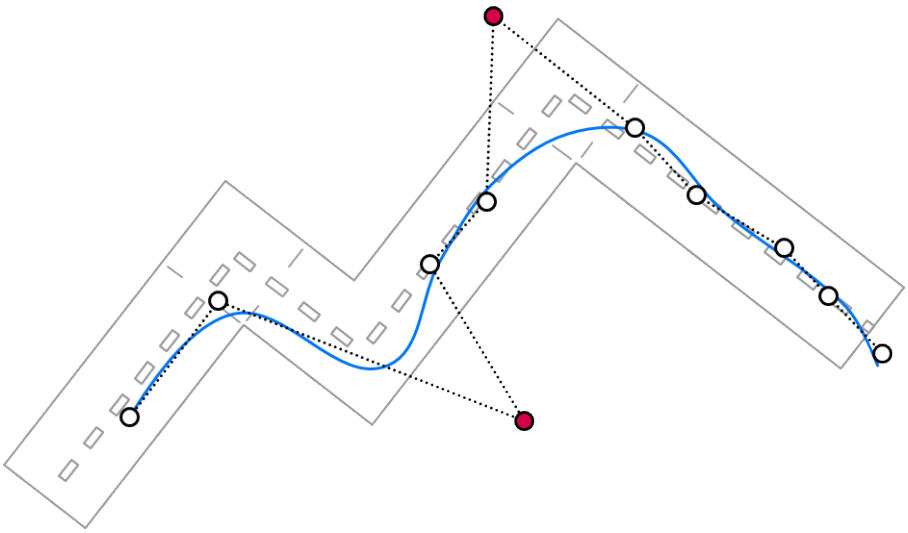


Figure 1.3: A moving entity has its location probed periodically to construct a trajectory (dotted) that accurately approximates the actual path of the entity (blue). The red probes lie very far away from the actual path and can thus be considered outliers.

effects on further processing of the trajectory data, so it is important that they are removed. See Figure 1.3. Determining which probes are outliers and which are not relies on having some sort of context for telling the two apart. For example, if we have a set of trajectories covering a similar route, but one of the trajectories has one probe clearly not on the route, it is likely to be an outlier. Or if we are working with car trajectories and have access to the underlying road network, we can assume that probes not lying on any road are likely to be outliers. Sometimes, however, these more comprehensive contexts are not available.

In this chapter, we aim to create an outlier detection approach that works for when only minimal context is available. We consider the scenario where only one trajectory is considered at a time, and the only available context is some idea of the physical properties, such as the maximum speed, of the entity being tracked. We give three optimal algorithms for finding the maximum subsequence of probes that are *consistent* according to the chosen physics model. I.e. we wish to find the largest set of probes such that the entity could have started at the earliest included probe, and then visited each of the measured locations at the measured times without violating

the physics model. Our results are output sensitive in the number of found outliers.

We give an $O(n \log n \log^2 k)$-time algorithm, where n is the number of probes in the trajectory and k the number of outliers, fora bound on the maximum velocity. For models where consistency is *concatenable*, meaning we can concatenate consistent subsequences where one ends where the other begins and always get a consistent subsequence, we give an $O(nk)$-time algorithm. For a non-concatenable model with a bound on velocity and acceleration, we give a $O(nk^2 \log k)$ time algorithm, under appropriate realism conditions.

To test our algorithms we run experiments on several real-world trajectory data sets. We compare our algorithms to previously existing algorithms and some greedy benchmark algorithms. We show that our approach is able to retain more of the probes of an input trajectory than the benchmarks. We observe that the change between a speed-bounded model and an acceleration-bounded model is not so big in practice. Finally, with a sensitivity analysis we show that the choice for the physics model parameters is very important for the performance of the algorithms.

This chapter presents work that was published in the paper *Maximum Physically Consistent Trajectories* [46].

Chapter 3: Simplification Curve simplification is a long-studied problem with many different applications. Since the rapid increase in GPS data, there has also been an increase in research into simplifying trajectories specifically, but since trajectories can also be seen as annotated polygonal curves, general curve simplification methods also work.

The aim of simplifying a curve is to reduce the complexity while still maintaining similarity to the original curve. To measure the similarity, a curve distance measure is used. In this chapter, we consider variations of this problem using six different curve distance measures. The most often studied cases of curve simplification look into selecting a subset of the vertices of the original curve. Since the simplified curve consists of the remaining vertices connected by straight line segments, when vertices get left out we can say that the simplified curve takes a *shortcut* past that vertex. The simplified curve can also shortcut past several vertices at once. In this case, the distance measure can be applied locally, only looking at the shortcutted section rather than the entire curve. This is the approach used in most curve simplification literature.

Van Kreveld et al. [89] have shown that when applying the curve distance measure globally, i.e. we only care about the distance between the full original and simplified curves, rather than caring about the distance between each individual shortcut and its corresponding original section, the complexity of the simplified

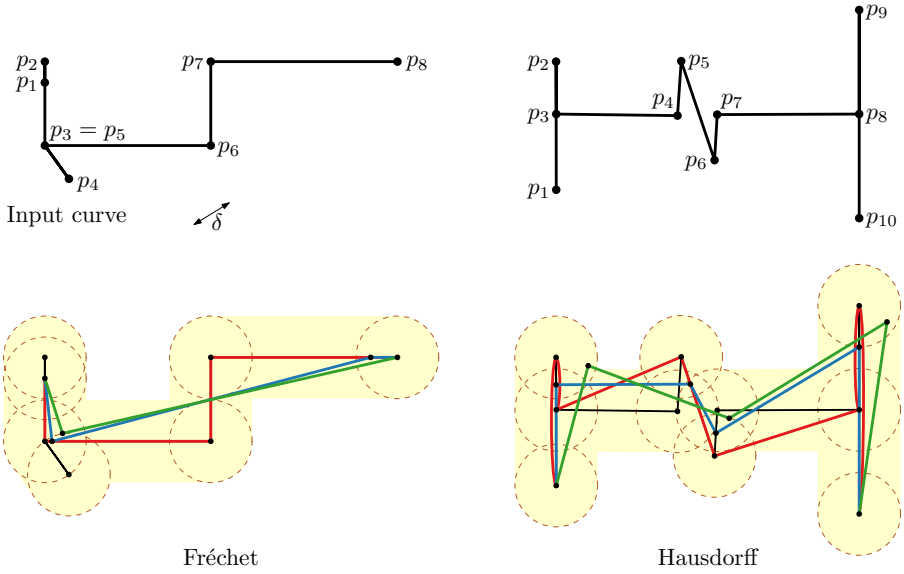


Figure 1.4: Different variants of global curve simplification. Red is the optimal vertex-restricted simplification, blue is the optimal curve-restricted simplification and green is the optimal non-restricted simplification.

curve can be lower than with the measure applied locally. Likewise, we can also let go of the restriction that the vertices of the simplified curve are vertices of the original curve. We can choose to only require that the simplified vertices lie somewhere on the curve, or we can only ask that the simplified curve starts and ends at the same points, but otherwise is just a polygonal curve. We refer to these restrictions on the problem as vertex-restricted, curve-restricted and non-restricted simplifications, respectively. See Figure 1.4. Curve- and non-restricted simplification, and especially the global application of the distance measure have received much less attention than local, vertex-restricted simplification has thus far. This has left several open questions on whether polynomial time algorithms exist for different cases, which we address in this chapter.

We give polynomial time algorithms for various variants and prove the NP-hardness of others. Systematically covering all of these variants shows a surprising pattern in that curve-restricted simplification seems to generally be harder than both vertex- and non-restricted versions of the problem.

This chapter presents work that was published in the papers *Global Curve Sim-*

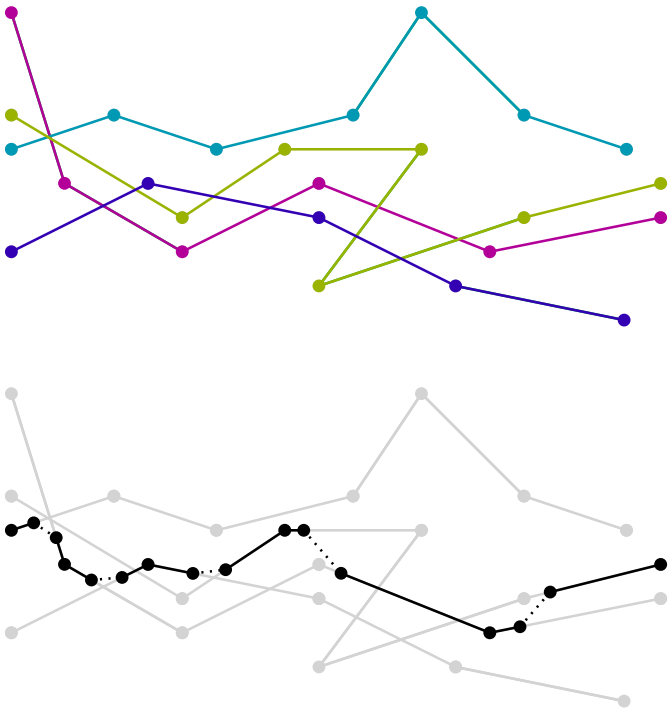


Figure 1.5: Top: A trajectory cluster, where no trajectory that can be picked is a good representative for the cluster. Bottom: A trajectoid constructed to be representative of the cluster. The dashed sections represent discontinuities where the trajectoid switches which trajectory is followed. This can only occur when the old and new entity are closer than a distance threshold at the time of the switch.

plification [86] and *Embedding Ray Intersection Graphs and Global Curve Simplification* [85].

Chapter 4: Representative Trajectories For large data sets, a clustering algorithm can be used to bring more structure, by dividing the data up into clusters. The aim is that data points will have more in common with the other data points in their cluster, than with data points outside of their cluster. For trajectory data, clustering can be very useful. The movement patterns sought after in trajectory data mining

are often restricted to a small area or timespan, so instead of looking at an entire data set at once it makes more sense to look at clusters of trajectories that are similar in location, time, and shape. To visualise a cluster, a representative trajectory can be chosen. This is a trajectory that captures the essence of the trajectories in the cluster somehow. A representative trajectory can either be chosen from the trajectories in the cluster, or one can be constructed.

One approach for constructing a representative trajectory is given in the paper Central Trajectories, by Van Kreveld et al. [90], which we have studied for this chapter. For a trajectory cluster whose probes fall within a common time interval, this approach constructs a *trajectoid*, a function that maps the time interval to different trajectories in the cluster. This can be thought of as a new trajectory constructed out of pieces of the trajectories of the cluster being summarized. See Figure 1.5. At the times where the trajectoid switches between pieces of different trajectories, there are *discontinuities*, where the endpoints of the different pieces do not overlap but have some distance between them. The maximum allowed distance ε is a parameter of the algorithm. The specific trajectoid that is constructed, the *central trajectory*, is the trajectoid that minimizes the distance between it and the center of the cluster.

To get a better picture of what the actual output complexity is when this algorithm is used in practice, we have created (to the best of our knowledge) the first implementation of this algorithm and applied it to several real-world data sets. We compare the output complexity to the theoretical worst-case complexity. Given that the output complexity could be much higher than the input complexity, we also study the effect of trajectory simplification as a pre- or post-processing step for the algorithm. We examine whether simplification is an effective way to keep the output complexity comparable to the input complexity, and if it is better to simplify before or after applying the algorithm.

We find the ratio of input to output complexity generally stays close to 1. We also find that simplifying the output is more impactful than simplifying the input in terms of reduced output complexity. Simplifying both reduces complexity even further in some cases, but the difference with simplifying only the central trajectory is small.

This chapter is based on an unpublished manuscript written in collaboration with Maarten Löffler and Roald Melssen.

Chapter 5: Road Network Generalization Many different applications require knowledge of a road network. The network is stored internally as an annotated graph. Depending on the context, we may want the full graph for a region, but for many cases the full graph contains too much detail. For example when visualizing

Figure 1.6: Top: Road network of the city of Leiden. Bottom: Heuristic generalization restricted to 30% of the total road length.

the road network of an entire country, drawing each small inner-city road for each city will make for a visualisation that is too cluttered. For those, and other cases, we want to use a generalized version of the graph. Like trajectory simplification, we want to preserve the important information while removing extraneous information. Here, we are looking to find a subgraph of the original road network that includes the most relevant edges (corresponding to the most important roads) and excludes less relevant edges. Approaches to estimate which areas of the graph should be kept already exist [137, 126, 123]. Since the rate of GPS data gathering has rapidly increased in recent years, this now also opens up the possibility of determining road importance by analyzing a data set of trajectory data that lies on the road network. For this to work, the trajectories are converted into paths on the graph first. For clarity, we will refer to these paths on the graph as *routes*. Edges in the road network will have a certain number of these routes traversing them. We will refer the number of different routes traversing a specific edge as the *ply* of that edge.

We introduce the *Route-preserving road network generalization* problem. We hy-

pothesize that solving this problem will give a good road network generalization that is data-driven. In this problem, we are given a graph G, a set of routes R and a budget B. We want to select a subset of the edges of G where the sum of edge lengths of the subset is at most B and the number of routes in R that are fully captured, i.e. all of their path edges are included in the subset, is maximized. The introduced problem focuses on fully capturing routes, instead of trying to maximize the total ply of the edges that are selected. This approach ensures commonly used routes are preserved in full in the generalization, and it helps prevent artifacts where only some segments of a longer road may be selected, leaving gaps in the road in the generalization. We conduct a theoretical study of this problem and show that it is unfortunately NP-hard for realistic cases.

While we show the problem cannot be solved exactly in a reasonable amount of time, we also investigate the quality of the solutions an approximation algorithm or a heuristical algorithm might produce, to see if good solutions to the route-preserving road network generalization problem are actually good generalizations. To do this, we define a score function that assigns a score to each route that we use for three different heuristical algorithms. For an example of a heuristical output, see Figure 1.6.

We compare these heuristical algorithms with a benchmark algorithm, that iteratively includes the edge with the highest ply into the subgraph until the budget is exhausted. We show that this benchmark approach indeed creates artifacts as we expected, that are not present using our heuristics.

This chapter presents work that was published in the paper *Route-preserving Road Network Generalization* [87].

Chapter 6: Conclusion In this chapter, we conclude by reviewing our contributions and giving an outlook of possible broad future directions of study.

Appendixes In Appendix A we describe the data sets that were used in our experiments. In Appendices B and C we go over the preprocessing steps that were applied to the data sets to prepare them for use in Chapters 4 and 5 respectively.

Chapter 2

Outlier Detection

2.1 Introduction

Before trajectory data can be used for anything, the data must first be collected. However, many ways to collect trajectories involve physical sensors which are prone to errors. For example, GPS readings notoriously stray far from their real location in urban canyons, resulting in trajectories with multiple significant outliers. These outliers pose problems for many analysis techniques such as clustering or grouping, and they skew the results of statistical methods. Hence, it is common practice to try to eliminate outliers as a *preprocessing task* before further data processing is done.

There are a variety of methods to remove outliers. Some techniques, such as smoothing or averaging the data, have a possibly negative impact on the complete trajectory. Others, such as map matching, are applicable only to trajectories that can be expected to coincide with a road network. This chapter focusses on *outlier detection*, that is, algorithms that identify outliers which are subsequently removed from the trajectory.

Specifically, we aim to identify outliers via the physical properties of the moving (real-world) entity. We consider two probes within a trajectory to be *consistent* for a particular physics model, if the corresponding entity could have traveled between the two measured locations in the time between the two probes. We present optimal algorithms to compute maximal consistent subtrajectories according to different (simplified) physics models. Before describing our results in more detail, we first introduce the necessary notation and formally state the problem.

Notation A trajectory T is obtained by tracking some entity. Let v^- be the minimum speed, or velocity, that the entity can achieve, and let v^+ be the maximum speed that the entity can achieve. Similarly, let a^- and a^+ be the minimum and maximum possible acceleration. These speed and acceleration bounds represent physical bounds, and thus the entity cannot exceed them at any time, even in between consecutive time stamps t_i and t_{i+1}. The actual continuous motion of an entity is assumed to be a continuous path $\pi : [t_1, t_n] \longrightarrow \mathbb{R}^d$ over time interval $[t_1, t_n]$ through d-dimensional space (typically, $d = 2$). We say that a path π *adheres to* the physics model if it never exceeds the bounds. For example, the speed is always in $[v^-, v^+]$ and the acceleration is always in $[a^-, a^+]$. A sequence of probes $T = \langle p_1, \ldots, p_n \rangle$ is *consistent* with the physics model, denoted $C(T)$, if and only if there exists at least one *witness*: a path $\pi : [t_1, t_n] \longrightarrow \mathbb{R}^d$ such that (i) for all $i \in \{1, \ldots, n\}$, $\pi(t_i)$ coincides with location p_i, and (ii) π adheres to the physics model. We sometimes write $C(p_i, p_j)$ instead of $C(\langle p_i, p_j \rangle)$.

Formal problem statement Given a trajectory T and a physics model, compute a maximum-size subsequence S of T such that S is consistent with the given model. When S has size ℓ, we consider the $k = n - \ell$ probes that were not included to be *outliers*.

Concatenability Regardless of the physics model, if a sequence T is consistent, then so is any subsequence S of T. But we cannot necessarily construct a consistent subsequence from smaller ones: the concatenation $\langle p_1, \ldots, p_n = q_1, \ldots, q_m \rangle$ of two consistent subsequences $T = \langle p_1, \ldots, p_n \rangle$ and $U = \langle q_1, \ldots, q_m \rangle$ with $p_n = q_1$ is not necessarily consistent. We call a physics model *concatenable* if such concatenations are always consistent under the model. An example of a concatenable model is one whose only limit is a maximum and minimum speed for the entity. Concatenable models generally allow more efficient algorithms.

Not all physics models are concatenable: for example, a model limiting both the speed and the acceleration is not concatenable. See Fig. 2.1 (left): both $T = \langle p_1, p_2 \rangle$ and $U = \langle p_2, p_3 \rangle$ are consistent, but $\langle p_1, p_2, p_3 \rangle$ is not. The main problem is that sequences U and T essentially require the entity to have two different speeds at p_2. The two subtrajectories U and T are concatenable in the acceleration-bounded model, under the condition that they have the same speed at their common probe. To capture this, we define a notion of *conditional consistency*, denoted $C(T \mid \gamma)$, in which a trajectory T is consistent, provided that it has a witness satisfying condition γ. In case $C(T \mid \gamma)$ and $C(U \mid \gamma')$ imply that the concatenation of T and U is consistent, we say that the physics model is *conditionally concatenable*. Hence, the model with

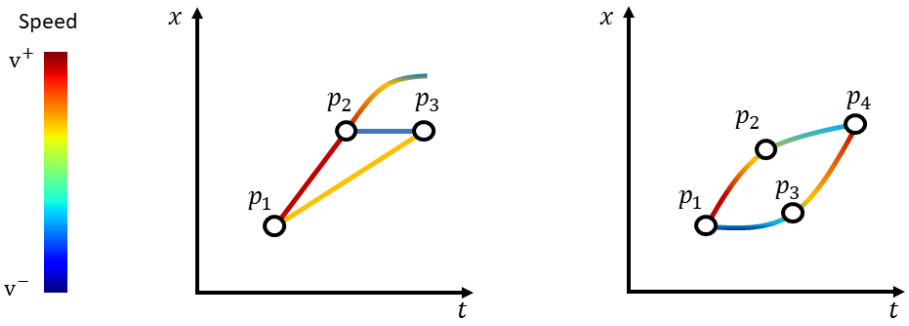


Figure 2.1: (Left) In an acceleration-bounded model $\langle p_1, p_2, p_3 \rangle$ is not consistent, even though $\langle p_1, p_2 \rangle$ and $\langle p_2, p_3 \rangle$ (and even $\langle p_1, p_3 \rangle$) are. Going from p_1 to p_2 requires the entity to have such high speed that it can't slow down quickly enough to reach p_3. (Right) A consistent subtrajectory through p_2 may require a different speed at p_4 than a subtrajectory that includes p_3.

bounded acceleration is conditionally concatenable, using the condition that the speed at the common probe is the same. The speeds attainable at a certain probe may depend on the subtrajectory so far, as illustrated in Fig. 2.1 (right).

Results and organization We present three algorithms and the results of computational experiments investigating the efficacy of our methods. Specifically, in Section 2.2 we describe a simple, optimal algorithm that runs in $O(nk)$ time for any concatenable physics model allowing constant time consistency checks between two probes. We then describe a more efficient algorithm which runs in $O(n \log n \log^2 k)$ time for the speed-bounded model in Section 2.3. Our final algorithm, described in Section 2.4, uses an acceleration-bounded model that can optionally also bound the speed. This algorithm runs in $O(nk^2 \log k)$ time under mild assumptions that are validated by our experiments. We also present a variant of this algorithm that introduces slack in the physics model to obtain an efficient approximate algorithm that achieves the given worst-case running time without assumptions.

In Section 2.5, we discuss the results of a series of computational experiments on real-world data using our open-source implementation. Specifically, we compare the quality of our algorithms to simple greedy approaches and conclude that our algorithms are more reliable, especially for trajectories with more than minor levels of noise. We also observe that the speed-bounded model approximates the acceleration-bounded model, though there is some dependency on the data set. Finally, we also briefly investigate how sensitive our results are to the model parameters: though

the speed bound is quite sensitive, the acceleration bounds have comparatively little influence on the number of outliers detected. We conclude with a discussion of our results in Section 2.6.

Related work For an overview of related work on outlier detection, see Section 1.1.

Physics models are often used in trajectory processing. Kalman filtering [94], for example, is based on a linear model for physical motion; its extensions handle more complex, nonlinear models. Note however, that Kalman filtering changes the probe positions rather than selecting a consistent subset. In a similar vein, physics models are used to reconstruct trajectories from data, replacing subtrajectories that cannot be physically realized with ones that can [109, 119]. Here, unrealistic subtrajectories are detected using a local time window, sliding over the trajectory.

Given a trajectory and physics model, we aim to determine the maximum number of probes that can be explained through a path adhering to the model. As such, our problem bears some resemblance to two other problems: computing a longest common subsequence (LCSS) and map matching. The former asks to compute the maximum subsequence of two strings [26] and has also been used to compute trajectory similarity [130]. Contrasting our approach, LCSS requires that both trajectories are known. For related work on map matching we refer to Section 1.1. Dealing with noise naturally arises in this application. Though we do not investigate this here, explicit outlier removal before map matching may improve results of simple and faster algorithms; postprocessing map matching results using our methodology may give rise to more realistic results. However, the primary difference is that our method does not rely on knowledge of the street network: the space of potential paths is defined implicitly and as such our methodology is more broadly applicable to movement that does not follow a predefined network (pedestrians, ships, airplanes). Map matching also does not tend to take physics into account, relying only on the positions of the probes.

▸ 2.2 Concatenable consistency model

We assume an arbitrary concatenable physics model that allows consistency checks between two probes in $O(f(n))$ time for some function f; typically, $f(n) = O(1)$. We follow the methodology of the Imai-Iri line-simplification algorithm [73]. Let $G = (V, A)$ be a directed acyclic graph with a vertex v_i for each probe p_i of T and an edge from v_i to v_j if and only if $C(p_i, p_j)$. This graph has $O(n^2)$ edges; each can be tested in $O(f(n))$ time. By concatenability, a path in G describes a consistent

subsequence. Since G is directed and acyclic, we compute a longest path in G, and thus a maximum consistent subsequence of T, in $O((|V| + |A|)f(n)) = O(n^2 f(n))$ time.

We now develop an output-sensitive variant of this algorithm. Rather than constructing the full graph, we build a subgraph G' in which each vertex v has at most one incoming edge (u_v, v). In particular, u_v and v's associated probes are the last probes of a longest consistent subsequence T_v ending in v's probe. Let ℓ_v denote the length of T_v.

Lemma 2.2.1. *Let T be a trajectory, M be a physics model that is concatenable, and $G' = (V, A)$ be a DAG consisting of one vertex v_i for each probe p_i of T and an edge from vertex u_v to vertex v if and only if v does not already have an incoming edge and the associated probes are consistent in M and are the last probes of a longest consistent subsequence T_v ending in v's probe. The probes associated with a longest path in G' are then a maximum physically consistent subsequence of T.*

Proof. Since M is concatenable, we know that if there is an edge from u_v to v, the probe associated with v must be consistent with each probe that is associated with an ancestor of u_v. Let v^* be the vertex associated with the last probe of a maximum physically consistent subsequence of T. By definition, v^* has a parent u_{v^*} such that their probes end a longest subsequence ending in v^* of length ℓ_{v^*}. Since v^* is part of a maximum physically consistent subsequence, ℓ_{v^*} has maximum length over all possible consistent subsequences of T. We now also know that there must be a consistent subsequence of length $\ell_{v^*} - 1$ ending in the probe associated with u_{v^*}. And so, u_{v_*} must also have a parent vertex whose probe is the endpoint of a consistent subsequence of length 1 less which also has a parent, and so on until we reach a vertex with no parent whose probe has no earlier consistent probes. All of the vertices of this backwards traversal of the graph form a longest path, and their associated vertices form a physically consistent subsequence of length ℓ_{v^*}, making it a maximum physically consistent subsequence of T. □

Now we will show how we can construct G', giving an output-sensitive algorithm that runs in $O(nkf(n))$ time, where n is the number of probes and k the number of outliers.

Theorem 2.2.2. *Consider a concatenable physics model that allows checking the consistency of a pair of probes in $O(f(n))$ time. A maximum consistent subsequence of a trajectory T with n probes can be computed in $O(nkf(n))$ time, where k is the number of outliers.*

Proof. We handle the probes in chronological order, maintaining a linked list $\mathcal{L}$ that stores, for each handled probe v, the value ℓ_v and the predecessor u_v in T_v. $\mathcal{L}$ is

ordered by the lengths ℓ_v in descending order. For a new probe w, we traverse $\mathcal{L}$ to find the first probe v consistent with w. Since $\mathcal{L}$ is ordered, we have thus found a longest consistent subsequence of length ℓ_v+1 ending in w. We now walk backwards in $\mathcal{L}$ and add w to the appropriate place in the list.

After we have handled all probes, the maximum consistent subsequence can be retrieved in $O(n-k)$ time, by starting at the head of the list and following the predecessor pointers. For each of the $n-k$ probes that end up in the longest subsequence, we perform one successful check preceded by at most k failed checks, and for the k outliers we perform at most n checks. This gives a total of $O(nkf(n))$ time performing the checks. The time for inserting a probe in $\mathcal{L}$ can be charged to the number of checks it performs: this takes $O(nk)$ time in total. Hence, the total running time for the algorithm is $O(nkf(n))$. □

As the speed-bounded model allows to check consistency between two probes in constant time, we thus obtain the following running time for this model. We can, however, improve upon this algorithm in case the trajectory has many outliers, as discussed in the next section.

Corollary 2.2.3. *For the speed-bounded model, a maximum consistent subsequence of a trajectory T with n probes can be computed in $O(nk)$ time, where k is the number of outliers.*

▸ 2.3 The speed-bounded model in 2D

We now consider the speed-bounded model, with maximum speed v^+, and trajectories in $\mathbb{R}^2$. We present an $O(n \log n \log^2 k)$-time algorithm to find a maximum consistent subtrajectory in this model. To this end we develop an insertion-only data structure that, given a probe q, can determine the length of a maximum consistent subsequence ending at q in $O(\log^3 n)$ time. Insertions are supported in $O(\log^3 n)$ time. By incrementally building the data structure in chronological order, we can determine the maximum consistent trajectory in $O(n \log^3 n)$ time. With a more careful analysis this can be improved to $O(n \log n \log^2 k)$ time.

▸ 2.3.1 A consistency data structure

Let P be a subset of probes from T, and let $\hat{t}$ be the time of the last probe in P. We develop a data structure $\mathcal{D}$ that can efficiently answer *consistency queries* on P. That is, for a given new query probe q occurring at time $t \geq \hat{t}$, we can test whether there is a probe in P consistent with q. We view the probes in P as points in $\mathbb{R}^3$, with the

third axis being time, that is, $p_i = (x_i, y_i, t_i)$. probes p_j, with $j > i$, that are consistent with p_i must lie inside a cone that starts at p_i and has radius $v^+(t - t_i)$ at time $t \geq t_i$; see Fig. 2.2. We call this cone the *reachable region* of p_i; testing whether p_j is in the reachable region of p_i takes $O(1)$ time.

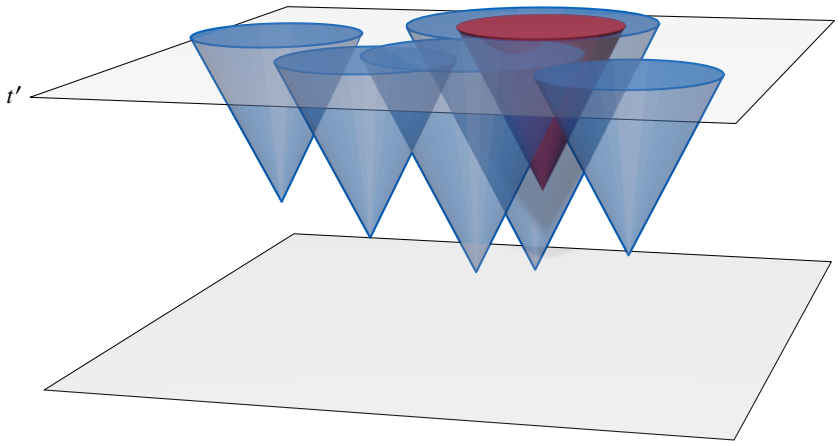


Figure 2.2: Each probe defines a reachable region (a cone), that intersects the plane at time t' in a disk. These disks define an AWVD. A probe p_j is consistent with an earlier probe p_h if (and only if) its cone (shown in red) is contained in the cone of p_h.

To determine if a probe q at time $t_q \geq \hat{t}$ is consistent with any probe of P we use an *additively weighted Voronoi diagram* (AWVD) [57, 52]. Given a set of disks with centers $\{v_1, \ldots, v_l\}$ and radii $\{r_1, \ldots, r_l\}$, this diagram partitions the plane into cells $\{c_1, \ldots, c_l\}$ associated with the disks, such that for any point $v \in c_i$ it holds that $d(v, v_i) - r_i \leq d(v, v_j) - r_j$, for all $v_j \neq v_i$. Here, d is a distance measure (in our case the Euclidean distance), and equality holds only on boundaries between cells. See Figure 2.3.

We construct an AWVD on the probes in P by using the locations as the centers and picking $r_i = v^+(t' - t_i)$ for every probe for some arbitrary $t' > t_n$. Since the disks all grow at the same rate as t' increases the precise value of t' does not matter as it will not cause the containment relations between disks to change. Observe that a probe p_j is consistent with p_h if the reachable region of p_j at t' is inside the reachable region of p_h at t' (see Fig. 2.2). We preprocess the AWVD for point location queries. Let $\mathcal{D}$ denote the resulting data structure, which we refer to as a *consistency data structure*. We can now query $\mathcal{D}$ with a new probe $q = p_q$, giving us

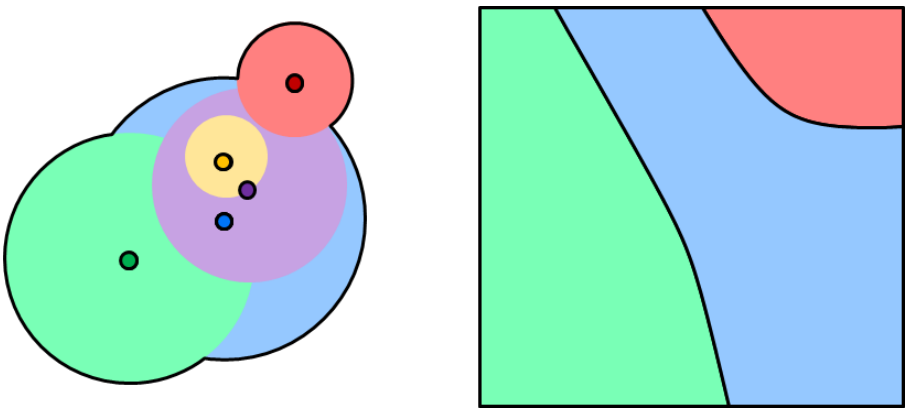


Figure 2.3: Left: a set of disks. Right: AWVD defined on the disks. If one disk is fully eclipsed by another, it does not have its own cell in the AWVD.

a previous probe p_c and a distance s_c between the disks (p_q, r_q) and (p_c, r_c), given by $s_c = d(p_q, p_c) - r_q - r_c$, where d measures the Euclidean distance between the points in the plane, that is, ignoring the temporal component. The following lemma then gives us that $\mathcal{D}$ can be used to answer consistency queries.

Lemma 2.3.1. *Let $\mathcal{D}$ be a consistency data structure on a set P of probes and let $q = p_q$ be a query probe, resulting in probe p_c on $\mathcal{D}$. If $s_c \leq -2r_q$ for the resulting distance s_c of p_c with p_q, then p_c is consistent with q. Otherwise, no probe in P is consistent with q.*

Proof. We denote $w_j = (x_j, y_j)$ for compactness. Let $p_q = (x_q, y_q, t_q)$ be the q-th probe in a trajectory and let $\mathcal{D}$ be the consistency data structure on a subset $P \subseteq \{p_1, \ldots, p_{q-1}\}$. Let p_c be the probe associated with the found cell in $\mathcal{D}$ containing (x_q, y_q) and let $s_c = d(w_q, w_c) - r_q - r_c$.

Since $s_c = d(p_q, p_c) - r_q - r_c$, we can rewrite the inequality $s_c \leq -2r_q$ to $d(p_q, p_c) \leq r_c - r_q$, Applying the definition of r_c and r_q, the right-hand side can be rewritten to $v^+(t' - t_c) - v^+(t' - t_q) = v^+(t_q - t_c)$. In other words, we have that the distance between the probes is at most the maximum distance that the object can travel in the given time span. Hence, p_q is consistent with p_c. In this case, p_q could also be consistent with other probes, but this does not matter for the purposes of the algorithm.

Analogously, if the given inequality does not hold, the distance between the two probes is larger than what the object can cover when traveling at maximum speed: they are not consistent.

To show that no other probe in P can be consistent in this case, observe that the

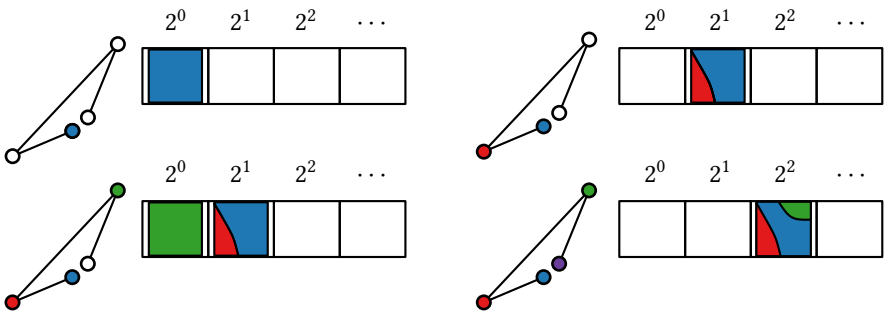


Figure 2.4: Inserting elements using Bentley-Saxe. The colored probes in the trajectory are the elements in the insertion-only consistency data structure, being inserted in the order blue,red,green,purple.

definition of the AWVD gives us that $d(p_q, p_c) - r_c \leq d(p_q, p_k) - r_k$ for all $p_k \in P \setminus \{p_c\}$. Subtracting r_q from both sides, we get that $d(p_q, p_c) - r_c - r_q = s_c \leq d(p_q, p_k) - r_k - r_q$. Thus, if $s_c > -2r_q$, we must also have that $d(p_k, p_c) > r_k - r_q$. Thus, all probes p_k are not consistent with probe p_q. □

We can construct the AWVD for a set of m probes and preprocess this AWVD for point-location queries in $O(m \log m)$ time [57, 52]. The resulting data structure uses $O(m)$ space, and can answer point-location queries in $O(\log m)$ time. Since a single consistency check takes constant time, we can also answer consistency queries in $O(\log m)$ time.

▸ 2.3.2 Supporting insertions

Next, we describe how to extend our consistency data structure to support insertions. Testing whether a probe is consistent with any previous probe of a subsequence of T is a decomposable search problem. Thus, we use the approach by Bentley and Saxe [22] to turn our consistency data structure into an efficient insertion-only data structure.

For a set of m probes, we maintain $O(\log m)$ instances of our static data structure $\mathcal{D}_1, \ldots, \mathcal{D}_{O(\log m)}$ (see Fig. 2.4). Every probe is in one of these $O(\log m)$ data structures. Data structure $\mathcal{D}_i$ has size 2^{i-1}. On insertion, we create a new $\mathcal{D}_1$ with the inserted probe. When we get two data structures of the same size 2^i, we remove both and replace them by a single data structure of size 2^{i+1}. We repeat this process until all data structures have a unique size. To answer a query we simply query all $O(\log m)$

data structures.

The above construction together with the consistency query structure gives $O(\log^2 m)$ time for a query and $O(\log^2 m)$ amortized time for an insertion. These bounds can be made worst case as well [116]. We summarize our results in the following lemma.

Lemma 2.3.2. *There is a consistency data structure $\mathcal{D}$ that can store a subset P of m probes from T and can answer consistency queries for query points q at time $t \geq \hat{t}$, in $O(\log^2 m)$ time, and supports insertions in $O(\log^2 m)$ time. Here, $\hat{t}$ denotes the time of the last probe currently in P. The data structure uses $O(m)$ space.*

▸ 2.3.3 Maximum subsequence queries

We now use the data structure from Lemma 2.3.2 to build a dynamic data structure that, for a new query probe $q = p_q$ can determine the length ℓ_q of a longest consistent subsequence $T_q \subseteq P$ ending at q. We store the probes in $p \in P$ in the leaves of a balanced binary tree $\mathcal{T}$, ordered by the length ℓ_p of the longest consistent subsequence ending in p. So the leftmost leaf has a probe that is not consistent with any earlier probe, and the rightmost leaf is associated with the probe that ends the longest consistent subsequence possible with the probes inserted thus far. Since multiple probes can end a subsequence of the same length this is a partial ordering. Each internal node v with right child r corresponds to a subset $P_r \subseteq P$, i.e. the probes stored in leaves in its right subtree. v stores the minimum ℓ_p, with $p \in P_r$, occurring in its right subtree, and a consistency data structure $\mathcal{D}_v$ built on the set P_r (see Fig. 2.5).

Given a query probe q, we find a probe $u \in P$ consistent with q with maximum length ℓ_u. It then follows that a maximum-length consistent subsequence T_q ending in q has length $\ell_u + 1$, and that u is the predecessor of q in T_q. To find u we start at the root v and query $\mathcal{D}_v$ to test whether any probe in the right subtree is consistent with q. If so, we repeat the process in the right child. If not, we move to the left child. This way we get the longest-path probe that is consistent with our query probe q.

To insert a new probe q, we find the leaf corresponding to length ℓ_q and insert q in the appropriate associated data structures of all ancestors along this root to leaf path. To keep the tree $\mathcal{T}$ balanced, we implement it using a BB[α] tree [27, 113]. When a subtree rooted at a node v becomes unbalanced, we rebuild it and its associated data structures from scratch.

With the lemma below, we prove that this data structure can be implemented efficiently.

Lemma 2.3.3. *There is a data structure $\mathcal{T}$ that can store a subset P of m probes from T and that can find, given a query probe q at time $t_q \geq \hat{t}$, (the length ℓ_q of) a longest*

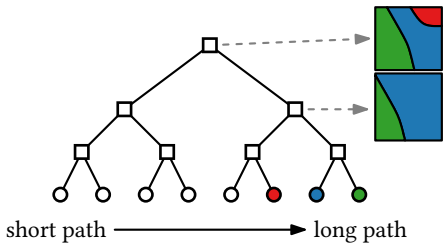


Figure 2.5: Data structure for maximum subsequence queries.

consistent subtrajectory T_q ending in q in $O(\log^3 m)$ time. The data structure uses $O(m \log m)$ space and supports insertions in $O(\log^3 m)$ amortized time. Here, $\hat{t}$ denotes the time of the last probe currently in P.

Proof. To answer a query we follow a path from the root down to a leaf, and query the associated data structure at each node. Each such query takes $O(\log^2 m)$ time (Lemma 2.3.2), and thus the total query time is $O(\log^3 m)$. Since each probe is stored in the associated data structure of $O(\log m)$ nodes, the total space use is $O(m \log m)$, and the total direct cost of an insertion is $O(\log^3 m)$. To prove the lemma, we need to bound the costs due to rebalancing operations.

Assume an insertion of a node triggers a rebalance operation for a subtree v of m_v elements, and let $C(m_v)$ be the total construction time: the time that it takes to acquire all elements in the subtree and construct a perfectly balanced tree (with its associated data structures). By the BB[α] definition, $(1 - 2\alpha)m_v - 2$ insertions must have occurred in v to trigger the rebalance operation. This implies that the amortized rebalance time per insertion is $\frac{C(m_v)}{(1-2\alpha)m_v-2}$.

Consider the tree after rebalancing. At height $h > 0$ we have intermediate nodes, each requiring a data structure $\mathcal{D}$ constructed on $O(2^h)$ elements. There are $O(m_v/2^h)$ nodes at height h. To construct nodes with height h, we require a total time complexity of $O((m_v/2^h)2^h \log^2(2^h)) = O(m_v h^2)$. Let $H = O(\log m_v)$ be the height of the entire subtree; summing up the construction time of all heights in the subtree gives the total construction time $C(m_v) = \sum_{h=1}^{H} O(m_v h^2) = O(m_v H^3) = O(m_v \log^3 m_v)$. Amortized, this construction cost and hence the entire insertion time is $O(\log^3 m)$.[1] □

[1]Note that in our improved bound in Theorem 2.3.5 the total reconstruction time $C(m_v)$ is simply $O(m_v \log m_v \log^2 k)$, as rebuilding the associated data structure of a node takes $O(m_v \log^2 k)$ time rather than $O(m_v \log^2 m_v)$ time.

▸ 2.3.4 Maximum consistent subtrajectories

To compute a maximum-length consistent subtrajectory of T, we process all probes in chronological order. For each we simply query the data structure described in Lemma 2.3.3, and then insert it. This results in an $O(n \log^3 n)$-time algorithm. Next, we show that we can improve this to $O(n \log n \log^2 k)$, where k is the number of outliers.

Lemma 2.3.4. *For two consistent probes p_i and p_j with $i < j$, the reachable region for p_j for all $t > t_j$ is contained in the reachable region of p_i.*

Proof. In the 3-dimensional space (with the third dimension being time), the reachable region of each probe is an upward cone starting at the probe, with slope v^+. As p_j is consistent with p_i, the former lies inside the latter's cone. As their direction and slope are the same, the cone for p_j is thus contained in the cone for p_i.

We can equally see this in 2-dimensional space. Consider an arbitrary time $t > t_j$. A hypothetical probe p^* at time t consistent with p_j must be within distance $\mathrm{v}^+(t - t_j)$. Since p_i and p_j are consistent, we know that their distance is at most $\mathrm{v}^+(t_j - t_i)$. Through triangle inequality, we thus know that the distance between p_i and p^* is at most $\mathrm{v}^+(t - t_j) + \mathrm{v}^+(t_j - t_i) = \mathrm{v}^+(t - t_i)$. This readily implies that p^* is consistent with p_i as well. □

From the definition of the AWVD, we know that if a disk c_1 is strictly inside another disk c_2, then c_1 will have an empty associated cell in the diagram. Combining this with Lemma 2.3.4 shows that any subset of $m \geq 1$ probes thus produces a diagram with at most $\min(m, k)$ cells. Hence, a static consistency data structure uses only $O(\min(m, k))$ space, and querying it requires $O(\log(\min(m, k)))$ time. When we insert a new probe p_j into our insertion-only data structure, we first query the data structure to decide whether p_j is consistent with some earlier measure p_i. If so, we simply discard p_j rather than inserting it; even when inserting additional points, the cell of p_i will contain that of p_j so the presence of p_j in the diagram can never change the path a query probe takes when descending through $\mathcal{T}$. The query and insertion time therefore both become $O(\log^2 \min(m, k))$.

It now follows that the associated data structure $\mathcal{D}_v$ of every node in $v \in \mathcal{T}$ has size only $O(\min(n_v, k))$, thus querying it requires only $O(\log^2 k)$ time, and thus $O(\log n \log^2 k)$ time in total. Similarly, inserting a new probe takes amortized $O(\log n \log^2 k)$ time.

Theorem 2.3.5. *Given a 2D trajectory T with n probes, of which k are outliers, we can compute a maximum consistent subsequence of T for the speed-bounded model in $O(n \log n \log^2 k)$ time.*

▸ 2.4 The acceleration-bounded model

We now consider 1D trajectories where each probe is of the form $p_i = (x_i, t_i)$. We assume a physics model where both velocity and acceleration are restricted. The velocity must lie in the range $[v^-, v^+]$ for constants v^-, v^+. Since the trajectory is one-dimensional, the velocity and acceleration are as well. In addition, the acceleration must lie in the range $[a^-, a^+]$ for constants a^-, a^+. For simplicity, we assume $a^+ \geq 0$ and $a^- \leq 0$ and refer to deceleration as acceleration with a negative value.

For this acceleration-bounded model, we can still test in constant time if two probes p_i and p_j are consistent: we can check if the distance between the two probes can be traveled using velocities that lie in the range $[v^-, v^+]$. If there exists a velocity at p_i such that the required velocity at p_j can be reached by accelerating, then the pair $\langle p_i, p_j \rangle$ is consistent. Recall, however, that a physics model that limits acceleration is not concatenable: there may be a triplet of probes $\langle p_1, p_2, p_3 \rangle$ for which the probes are pairwise consistent, but the entire sequence is not (see Fig. 2.1 (left) for an example). Hence, we cannot use the algorithm described in Section 2.3.

In Section 2.4.1 we describe a dynamic programming algorithm which explicitly computes the velocities achievable at every probe and, using these velocities, finds the length of a maximum-length consistent subtrajectory. In Sections 2.4.2 and 2.4.3 we give additional details on the algorithm: In Section 2.4.2 we show how velocity intervals can be propagated between cells in the dynamic program. In Section 2.4.3 we define the order in which we compute our dynamic program cells which saves us from computing cells we can infer will be empty. In Section 2.4.4 we show how to retrieve the actual consistent trajectory. The running time of the dynamic program and of the retrieval procedure depends on the maximal fragmentation of the velocity intervals which can arise during the DP. In Section 2.4.5 we first argue that this number can be as large as $\Omega(n)$ for a linear number of sets of velocities. It is easy to see that the maximal fragmentation is at most $O(2^n)$, however, it is unlikely that this bound would ever be reached in practice. In the following we consider an acceleration-bounded model with some slack in the acceleration bounds, modeling real-world imprecision. This slack allows us to prove a linear upper bound for the fragmentation of any set of velocities. Finally, in Section 2.4.6 we explain how we extend the acceleration-bounded model to dimensions greater than one.

▸ 2.4.1 Computing the maximum length of a physically consistent subtrajectory

A subtrajectory T is generally not concatenable with another subtrajectory T' under the acceleration-bounded model, but is conditionally concatenable with T' when the velocities at probes that they have in common are the same (see Section 2.1). Intuitively, this follows from the fact that a bound on the acceleration prevents (discontinuous) jumps in velocity. Based on this, we observe the following:

Observation 1. *A (sub)trajectory $S = \langle p_1, \ldots, p_m \rangle$ is consistent in the acceleration-bounded model if and only if there are velocities $\langle v_1, \ldots, v_m \rangle$ such that for all $i \in \{1, \ldots, m-1\}$ we have that $C(p_i, p_{i+1} \mid \mathrm{v}_i = v_i, \mathrm{v}_{i+1} = v_{i+1})$.*

Observation 1 implies that our problem has an optimal substructure. Suppose we have found all subtrajectories of some length ℓ that are consistent. If we now want to know whether a subtrajectory of length $\ell + 1$ exists, we have to determine only whether there is a probe p' such that the observation holds for one of the subtrajectories when we add p' at the end. That is, there should be witness paths for both the subtrajectory and the trajectory between the last probe of the subtrajectory and p', that have a common velocity at the last probe of the subtrajectory. Hence, we can apply the dynamic programming paradigm to find the optimal length for which a subtrajectory is physically consistent.

More formally, for each probe p_i and each possible length $\ell \in \{1, \ldots, n\}$, we maintain the set of velocities $\mathcal{I}(\ell, i)$ such that for every velocity $v \in \mathcal{I}(\ell, i)$, a subsequence $S = \langle \ldots, p_i \rangle$ ending at p_i of length ℓ exists that is physically consistent and has velocity v at p_i, so that $C(S \mid \mathrm{v}_i = v)$. Let ℓ^* be the maximum value, such that a probe p_i exists for which the set $\mathcal{I}(\ell^*, i)$ is non-empty. It follows that the maximum consistent subtrajectory of T has length ℓ^*.

Given the set of possible velocities $\mathcal{I}(\ell, h)$ at p_h, we can then determine whether a consistent subsequence of length $\ell + 1$ exists that ends at a later probe p_i by using the conditional concatenability property: if we find velocities $v_h \in \mathcal{I}(\ell, h)$ and $v \in [\mathrm{v}^-, \mathrm{v}^+]$ such that $C(p_h, p_i \mid \mathrm{v}_h = v_h \wedge \mathrm{v}_i = v)$, then a consistent subsequence $\langle \ldots, p_h, p_i \rangle$ of length $\ell + 1$ exists. Hence, we obtain the following recurrence for $\mathcal{I}(\ell, i)$.

$$\mathcal{I}(\ell, i) = \begin{cases} \emptyset, & i < \ell \\ \{v \mid \exists h : C(p_h, p_i \mid \mathrm{v}_i = v)\}, & \ell = 2 \\ \{v \mid \exists h \exists \mathrm{v}_h : C(p_h, p_i \mid \mathrm{v}_h \in \mathcal{I}(\ell - 1, h), \mathrm{v}_i = v)\}, & \ell > 2 \end{cases}$$

Moreover, we prove in Lemma 2.4.1 below, that when the entity directly travels

from p_i to p_j, and leaves p_i with velocity v_i, the possible velocities with which it can arrive at p_j form a connected interval. It follows that the sets $\mathcal{I}(\ell, i)$ are actually sets of intervals.

Lemma 2.4.1. *Let p_i and p_j be probes with $t_i < t_j$, and let $v_1 \leq v \leq v_2$ be velocities. If $C(p_i, p_j \mid \mathrm{v}_j = v_1)$ and $C(p_i, p_j \mid \mathrm{v}_j = v_2)$, then we also have $C(p_i, p_j \mid \mathrm{v}_j = v)$.*

Proof. $C(p_i, p_j \mid \mathrm{v}_j = v_1)$ and $C(p_i, p_j \mid \mathrm{v}_j = v_2)$ imply that there are two witnesses: paths $\pi_1(t)$ and $\pi_2(t)$ between p_i and p_j that travel $\Delta x = x_j - x_i$ distance, obey the physics model and have velocity v_1 respectively v_2 at p_j. Let $a_1(t)$ and $a_2(t)$ denote the acceleration functions describing these paths.

The traveled distance Δx between t_i and t_j using any acceleration function $\hat{a}(t)$ and velocity v' at p_j is given by

$$\Delta x = (t_j - t_i)\left(v' - \int_{t_i}^{t_j} \hat{a}(t)\mathrm{d}t\right) + \int_{t_i}^{t_j}\int_{t_i}^{t} \hat{a}(t')\mathrm{d}t'\mathrm{d}t \tag{2.1}$$

Any new path π^* which we create using convex combinations $v = \beta v_1 + (1-\beta)v_2$ and $a(t) = \beta a_1(t) + (1-\beta)a_2(t)$ for $\beta \in [0,1]$, travels exactly the same distance by linearity of the integrals. Since π^* was created via convex combinations, we also know that it satisfies the velocity and acceleration constraints, since its velocity and acceleration always lie between the original velocities and accelerations at any time t in $[t_i, t_j]$. Hence, π^* is a witness that implies $C(p_i, p_j \mid \mathrm{v}_j = v_1 + (1-\beta)v_2)$ for any $\beta \in [0,1]$. □

Lemma 2.4.2 below shows how to propagate a single speed interval from p_i to p_j in constant time. The problem is clearly computable, and has $O(1)$ input complexity: two probes and a single interval of velocities. As such, the lemma readily follows. The precise propagation function we will derive in the next subsection.

Lemma 2.4.2. *Let p_i and p_j be two probes with $i < j$, and let I be an interval of velocities at p_i. The interval $I' = \{v \mid C(p_i, p_j \mid \mathrm{v}_i \in I \wedge \mathrm{v}_j = v)\}$ of achievable velocities can be computed in $O(1)$ time.*

▸ 2.4.2 Propagating a speed interval in $O(1)$ time

We need to determine the minimum and maximum speed for which the moving entity can arrive at a probe p_j, when starting at a probe p_i with some given velocity. We will show how to precisely compute this interval, providing the exact formula for propagation and thus proving that this is indeed computable. We do so for the minimum velocity; computing the maximum velocity is symmetrical. This minimum

velocity $v_{min} \in \mathbb{R}$ is the smallest value such that $C(p_i, p_j \mid v_i = v, v_j = v_{min})$ for two probes p_i, p_j and some initial $v \in \mathbb{R}$. Note that these velocities may be negative, indicating the direction of movement in the 1D space.

We need particular behaviors of the moving entity to accomplish this minimum velocity. These behaviors are determined by the travel time $\Delta t = t_j - t_i$ and distance $\Delta x = \|p_j - p_i\|$ between p_i and p_j as well as the initial velocity v: we want to travel the distance between p_i and p_j within the given time, while minimizing the velocity at p_j.

Checking consistency First, we determine whether *any* velocity can be obtained, that is, whether it is physically possible to travel distance Δx in Δt time, starting with velocity v. That is, we must test whether $C(p_i, p_j \mid v_i = v)$ holds. The maximum distance Δx^+ that can be traveled, is obtained by accelerating until reaching the maximum velocity v^+ and then maintaining that speed. Accelerating to maximum speed takes $t^+ = \frac{v^+ - v}{a^+}$ time. If $t^+ < \Delta t$, the maximum velocity is achieved and we can express Δx^+ as $(v + v^+)t^+/2 + (\Delta t - t^+)v^+$. Otherwise, this behavior accelerates maximally for the entire duration, in which case $\Delta x^+ = (2v + a^+\Delta t)\Delta t/2$. Analogously, we find an expression for the minimum distance Δx^- that can be traveled. As we can use a convex combination to achieve any traveled distance between these two extremes, we know that $C(p_i, p_j \mid v_i = v)$ if and only if $\Delta x^- \leq \Delta x \leq \Delta x^+$.

Note that if the above test indicates consistency, then we can look for a minimal (and maximal) velocity. If this consistency does not hold, we know that no velocity can be reached at all. For the remainder, we assume that $C(p_i, p_j \mid v_i = v)$ is indeed true.

Finding the minimum velocity We now strive to find the necessary behavior that results in the minimum velocity, v_{min}, assuming we have already confirmed the basic consistency described above. To visualize this behavior, we look at the velocity-time diagrams for the moving entity; see the examples in Fig. 2.6. The curve in this diagram represents the velocity as a function of time. In this diagram, we need that the total area under the curve is exactly the traveled distance Δx. The speed bounds v^-, v^+ are now represented as allowed minimum and maximum values for the curve. The bounds on the acceleration, a^-, a^+ translate to the minimum and maximum slope the curve can have at any time.

We can distinguish a number of situations where we need different behavior to get to the minimum velocity, depending on whether we travel at maximal velocity intermediately. If we can reach p_j by first maximally accelerating for some time t_{acc} and then maximally decelerating the rest of the time, this gives us the lowest possible

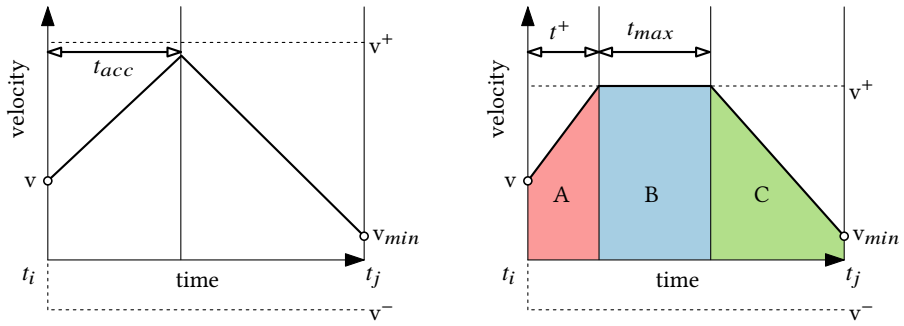


Figure 2.6: (Left) Velocity-time diagram for the behavior where one maximally accelerates for a time t_{acc} and then maximally decelerates to obtain the lowest possible speed v_{min}. (Right) Velocity-time diagram for the behavior of accelerating to the maximum velocity v^+ in time t^+, then maintaining this velocity for t_{max} time, and finally maximally decelerating to get the minimum velocity. Since v^- is assumed to be negative it is drawn below the time axis.

velocity (left diagram in Fig. 2.6). We can, however, encounter the case where the maximum velocity we reach with this behavior exceeds v^+. In this case, we can accomplish the minimum velocity by accelerating to v^+ in time t^+, retaining this speed for some time t_{max} and then maximally decelerating (right diagram in Fig. 2.6) for the remaining time. However, if the result of the appropriate situation above violates the minimum velocity bound v^-, then we can conclude that $v_{min} = v^-$. Now we will give lemmas to show that the first two behaviors indeed give the minimum velocity.

Lemma 2.4.3. *For consistent probes p_i and p_j, with a fixed v_i, the behavior of the entity moving from p_i to p_j, covering exactly Δx distance in Δt time that minimizes v_j, provided this does not exceed the velocity bounds of the physics model, is first maximally accelerating and then maximally decelerating. The length of time spent accelerating before switching to deceleration is chosen based on what is needed to cover exactly Δx.*

Proof. We now show that the behavior of maximally accelerating, followed by maximally decelerating indeed gives the minimum velocity, provided that the velocity bounds v^-, v^+ are never exceeded during this behavior. Without loss of generality, we further assume that $t_i = 0$ to simplify the exposition, and thus $t_j = \Delta t$.

The equation of 1D motion (Equation 2.12) must satisfied for $a(t)$, with additional constraints that $a(t) \in [a^-, a^+]$ for any $t \in [0, \Delta t]$. We then want to minimize the velocity at p_j, as given by Equation 2.13.

We represent the function $a(t)$ as follows:

$$a(\phi, t) = a^- + (a^+ - a^-)\phi(t) = a^- + \Delta a\psi(t) \tag{2.2}$$

where $\phi : [0, \Delta t] \longrightarrow [0, 1]$. This way, the acceleration bounds are trivially satisfied by the function.

Let $a(\psi, t)$ be the function representing maximal acceleration up to some time $t_{acc} \in [0, \Delta t]$ and then maximum deceleration until t_j. In addition, assume that the traveled distance is satisfied by this function. We can represent $\psi(t)$ by

$$\psi(t) = \begin{cases} 1 & t \leq t_{acc} \\ 0 & t > t_{acc} \end{cases} \tag{2.3}$$

Let $\psi'(t)$ be a function given by $\psi(t) + \delta\psi(t)$ where $\delta\psi(t)$ is a perturbation on the function, such that $a(\psi', t)$ still travels the required distance, but differs in at least one value t from $a(\psi, t)$. We now show that the velocity at p_j for this perturbed function is always greater than the velocity produced by $\psi(t)$.

By assumption, both travel the same Δx distance in Δt time. Thus, filling in Equation 2.12 for both gives us the following equality and its simplification:

$$\mathrm{v}\Delta t + \int_0^{\Delta t}\int_0^t (a^- + \Delta a\psi(t'))\mathrm{d}t'\mathrm{d}t = \mathrm{v}\Delta t + \int_0^{\Delta t}\int_0^t (a^- + \Delta a(\psi(t') + \delta\psi(t'))\mathrm{d}t'\mathrm{d}t \tag{2.4}$$

$$\int_0^{\Delta t}\int_0^t \delta\psi(t')\mathrm{d}t'\mathrm{d}t = 0 \tag{2.5}$$

We know that the velocity at p_j is given by Equation 2.13. We can now look at the difference $\Delta\mathrm{v}$ between this velocity for $\psi'(t)$ and for $\psi(t)$. This difference is given by

$$\Delta\mathrm{v} = \Delta a \int_0^{\Delta t} \delta\psi(t)\mathrm{d}t \tag{2.6}$$

Now, if the minimum velocity at p_j for $a(\psi', t)$ is smaller than the minimum velocity for $a(\psi, t)$, this would imply that $\Delta\mathrm{v}$ is negative for the corresponding $\delta\psi$ function.

By definition, the value of $\delta\psi(t)$ is non-positive for $t < t_{acc}$ and non-negative for $t > t_{acc}$, as otherwise the acceleration bounds would be violated. Equation 2.5 implies that $\int_0^{t_{acc}} \delta\psi(t)\mathrm{d}t < 0$ and $\int_{t_{acc}}^{\Delta t} \delta\psi(t)\mathrm{d}t > 0$. Equivalently, the integral $\int_0^t \delta\psi(t)\mathrm{d}t$ is non-increasing for the interval $[0, t_{acc}]$ and non-decreasing for the interval $[t_{acc}, \Delta t]$.

Assume for a contradiction that $\Delta\mathrm{v}$ is negative for some $\delta\psi$, that is, $\int_0^{\Delta t} \delta\psi(t)\mathrm{d}t < 0$. This automatically implies that $\int_0^t \delta\psi(t')\mathrm{d}t' \leq 0$ for any t in the interval, due to

the non-decreasing and non-increasing properties of the integration interval. But then the integral of Equation 2.5 is by definition negative, which means that $a(\psi', t)$ does not travel Δx distance. Hence, we must have that $\Delta v \geq 0$. Observe that $\Delta v = 0$ only if $\delta\psi(t)$ is zero.

To prove that maximally decelerating and then accelerating results in the maximum velocity follows a similar argumentation. □

Lemma 2.4.4. *For consistent probes p_i and p_j, with a fixed v_i, the behavior of the entity moving from p_i to p_j, covering exactly Δx distance in Δt time that minimizes v_j, provided the behavior from Lemma 2.4.3 exceeds the velocity bounds of the physics model, is first maximally accelerating to v^+, maintaining this speed for some time, and then maximally decelerating. The length of time spent at v^+ before switching to deceleration is chosen based on what is needed to cover exactly Δx.*

Proof. We now prove that, if we maximally accelerate to the velocity bound v^+ in time t^+, retain this speed for time t_{max}, and then maximally decelerate, this indeed gives us the lowest possible velocity, if the previous situation does not apply – that is, $t^+ \leq t_{acc}$, and we never reach the velocity lower bound v^-. Without loss of generality, we assume that $t_i = 0$ and $t_j = \Delta t$.

We follow the argumentation as described in the previous proof. We again describe the acceleration behavior using $a(\psi, t)$ for a to be defined function $\psi(t)$. We assume that $a(\psi, t)$ travels the required distance given the initial velocity. But now, the behavior of this case yields a slightly different function for ψ, do describe $a(\psi, t)$:

$$\psi(t) = \begin{cases} 1, & t \leq t^+ \\ \alpha, & t^+ < t \leq t^+ + t_{max} \\ 0, & t > t^+ + t_{max} \end{cases} \tag{2.7}$$

Here, $\alpha = -\frac{a^-}{\Delta a}$ indicates an acceleration of zero. For brevity, we call the three time regions with the different behaviors A, B and C, see Fig. 2.6.

We again perturb $\psi(t)$ to get a function $\psi'(t) = \psi(t) + \delta\psi(t)$. Here, we again assume that $\delta\psi(t)$ is not the zero function. To obey the acceleration bounds, we observe that $\delta\psi(t)$ is non-positive in region A and non-negative in region C, which implies that $\int_0^t \delta\psi(t)\mathrm{d}t$ is non-increasing in region A and non-decreasing in region C.

For region B, we observe that $\psi(t)$ describes the behavior that results in the highest possible velocity in that region. So, the velocity at a time t in region B that results from $\psi'(t)$ can be at most the velocity obtained via $\psi(t)$. The velocity at any

time t is given by

$$\mathrm{v}(\phi, t) = \mathrm{v} + ta^{-} + \Delta a \int_0^t \phi(t)\mathrm{d}t \tag{2.8}$$

for acceleration function $a(\phi, t)$. Thus, we can now formalize the above observation as $\mathrm{v}(\psi', t) \leq \mathrm{v}(\psi, t)$ for all $t \in B$. Using the definition of $\mathrm{v}(\phi, t)$ and simplifying, we obtain that

$$\int_0^t \delta\psi(t) \leq 0 \tag{2.9}$$

should hold for all $t \in B$. Since we want that ψ and ψ' travel the same distance Δx in Δt time, we again get the identity

$$\int_{t_i}^{t_j} \int_{t_i}^{t} \delta\psi(t)\mathrm{d}t'\mathrm{d}t = 0 \tag{2.10}$$

as was shown in the proof for Lemma 2.4.3. Similar to the argumentation in the proof for Lemma 2.4.3, we look at the difference in minimum speed $\Delta\mathrm{v}$ at t_j:

$$\Delta\mathrm{v} = \Delta a \int_{t_i}^{t_j} \delta\psi(t)\mathrm{d}t \tag{2.11}$$

Again, $\psi'(t)$ has a lower minimum velocity if $\Delta\mathrm{v}$ is negative. For this to happen, $\int_{t_i}^{t_j} \delta\psi(t)\mathrm{d}t$ has to be negative.

Now, assume for a contradiction that for some $\delta\psi$, $\Delta\mathrm{v}$ is less than zero, such that the minimum velocity using $a(\psi', t)$ is less than that from $a(\psi, t)$. From Equation 2.9, we see that $\int_0^t \delta\psi(t)\mathrm{d}t$ is non-positive for all t in regions A and B. In particular, at the end of region B, the integral is non-positive. We distinguish two cases.

The integral is zero Suppose the integral $\int_0^t \delta\psi(t)\mathrm{d}t$ is zero at the end of region B. Then, since the integral is non-decreasing in region C as established before, we cannot have that $\int_0^{\Delta t} \delta\psi(t)\mathrm{d}t$ is less than zero: $\Delta\mathrm{v}$ is non-negative, which gives a contradiction.

The integral is negative Suppose now that $\int_0^t \delta\psi(t)\mathrm{d}t$ is negative at the end of region B. If we want $\Delta\mathrm{v}$ to be negative, this requires that the $\int_0^{\Delta t} \delta\psi(t)\mathrm{d}t$ is negative. Since the integral is non-decreasing in region C, we must have that the integral is negative everywhere in region C to accomplish this. But then the traveled distance for $\psi(t)$ and $\psi'(t)$ is not the same, since Equation 2.10 is less than zero. This again gives a contradiction.

From the previous argumentation, we can conclude that for any choice of $\psi'(t)$

that satisfies the traveled distance requirement, the minimum velocity at p_j is at least the minimum velocity obtained by using $\psi(t)$. □

Now that we have shown that these two cases are optimal (unless the minimum velocity is v^-), we can compute what the minimum velocity actually is. To compute the minimum velocity for the first and second case, we will use the equation of motion in 1D, given by

$$\Delta x = v\Delta t + \int_{t_i}^{t_j} \int_{t_i}^{t} a(t')\mathrm{d}t'\mathrm{d}t \tag{2.12}$$

This describes the aforementioned requirement that the area under the curve in the velocity-time diagram is the distance Δx between p_i and p_j. To find the minimum velocity for the first two described situations, we fill in the shape for the acceleration $a(t)$ and determine the minimum velocity, given by

$$v_{min} = v + \int_{t_i}^{t_j} a(t)\mathrm{d}t \tag{2.13}$$

Maximally accelerate, then maximally decelerate We first consider the situation where we do not reach the velocity bound when maximally accelerating. In this first situation, our acceleration function is equal to

$$a(t) = \begin{cases} a^+, & t_i \leq t \leq t_i + t_{acc} \\ a^-, & t_i + t_{acc} < t \leq t_j \end{cases} \tag{2.14}$$

What remains is to determine t_{acc}. We do this by solving the 1D equation of motion (Equation 2.12) for the distance Δx between the probes. We fill in the acceleration function and integrate to get

$$\Delta x = v\Delta t + \frac{1}{2}a^+ t_{acc}^2 + (v + a^+ t_{acc})(\Delta t - t_{acc}) + \frac{1}{2}a^-(\Delta t - t_{acc})^2 \tag{2.15}$$

We can now solve this quadratic equation for t_{acc}. To simplify notation, we use $\Delta a = a^+ - a^-$ and $\bar{v} = \Delta x/\Delta t$, that is, the average required velocity to travel the distance in the given amount of time. We pick the root of the solution such that the resulting t_{acc} is in $[t_i, t_j]$ and get

$$t_{acc} = \Delta t - \sqrt{\frac{\Delta t}{\Delta a}}\sqrt{a^+\Delta t + 2(v - \bar{v})} \tag{2.16}$$

With this value, we can now determine the minimum velocity. We fill in Equation 2.13 and get

$$\begin{aligned} \mathrm{v}_{min} =& \mathrm{v}(t_i + \Delta t) = \mathrm{v} + t_{acc} a^+ + (\Delta t - t_{acc}) a^- \\ =& \mathrm{v} + \Delta t a^+ - \sqrt{\Delta a \Delta t} \sqrt{a^+ \Delta t + 2(\mathrm{v} - \bar{\mathrm{v}})} \end{aligned} \tag{2.17}$$

Note that this situation applies only if we do not exceed the velocity bounds when accelerating and decelerating. So we require that

$$\mathrm{v} + a^+ t_{acc} \leq \mathrm{v}^+, \qquad \mathrm{v}_{min} = \mathrm{v} + t_{acc} a^+ + (\Delta t - t_{acc}) a^- \geq \mathrm{v}^- \tag{2.18}$$

Accelerating to maximum velocity We now consider the situation where we reach the speed bound v^+. We accelerate for some t^+ time, until we are moving with velocity v^+, then we retain that velocity for some time t_{max}, and finally we maximally decelerate to get the minimum velocity. We can describe this behavior with the following acceleration function:

$$a(t) = \begin{cases} a^+, & t_i \leq t \leq t_i + t^+ \\ 0, & t_i + t^+ < t \leq t_i + t^+ + t_{max} \\ a^-, & t_i + t^+ + t_{max} < t \leq t_j \end{cases} \tag{2.19}$$

We now need to determine t^+ and t_{max}. As before, $t^+ = \frac{\mathrm{v}^+ - \mathrm{v}}{a^+}$ indicates the time needed to accelerate from v to v^+.

With the equation for t^+, we can now determine t_{max} by again solving the 1D equation of motion in Equation 2.12 with our new acceleration function. This gives us the following equation of motion, and solution for t_{max}:

$$\Delta x = \mathrm{v} t^+ + \frac{1}{2} a^+ (t^+)^2 + \mathrm{v}^+ (\Delta t - t^+) + \frac{1}{2} a^- (\Delta t - t_{max} - t^+)^2 \tag{2.20}$$

$$t_{max} = \Delta t - t^+ - \sqrt{\frac{3a^+}{a^-}(t^+)^2 + \frac{2\Delta t(\bar{\mathrm{v}} - \mathrm{v}^+)}{a^-}} \tag{2.21}$$

Using the above, we now find the minimum velocity, by filling in Equation 2.13:

$$\mathrm{v}_{min} = \mathrm{v}^+ + (\Delta t - t^+ - t_{max}) a^- = \mathrm{v}^+ + \sqrt{\frac{3a^+}{a^-}(t^+)^2 + \frac{2\Delta t(\bar{\mathrm{v}} - \mathrm{v}^+)}{a^-}} a^- \tag{2.22}$$

This case is applicable only if $t_{max} > 0$ and the resulting $\mathrm{v}_{min} \geq \mathrm{v}^-$.

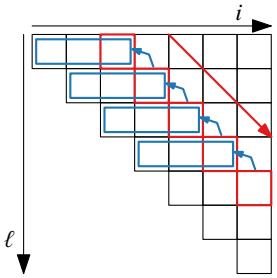


Figure 2.7: The order for computing $\mathcal{I}(\ell, i)$.

Achieving v^- The extreme behavior of the previous two cases achieve the lowest possible speed, without violating v^+, a^+ or a^-. We can readily choose between the two cases, by comparing t_{acc} with t^+: if the former is at most the latter, the first case applies, and otherwise the second. However, the result may still violate the physics model, but only v^-. That is, if the computed v_{min} is below v^-. Our claim is that, in such a case, v_{min} is actually equal to v^-. Intuitively, the previous cases in fact achieve a velocity that is too low: we thus have slack to use less extreme behavior intermittently, such as standing still ($v = 0$) for a certain time.

Consider the behavior of the previous cases. If we follow the behavior but maintain the minimal velocity bound, we have too much area under the curve: we overshoot our traveled distance. We can compensate for this by accelerating less extremely or maintaining a velocity below v^+. Since some velocity is obtainable, we know that there is sufficient slack and can indeed achieve v^-, if the previous cases would violate the velocity bound of v^-.

▸ 2.4.3 Additional details on the dynamic program

Let now $\delta(\ell, i)$ denote the number of intervals in $\mathcal{I}(\ell, i)$. We refer to $\delta(\ell, i)$ as the *fragmentation* of $\mathcal{I}(\ell, i)$. Let $\delta_{\max}$ be the maximum fragmentation over all ℓ and i. Using the recurrence for $\mathcal{I}$ defined earlier, we can compute all values $\mathcal{I}(\ell, i)$ using dynamic programming. We compute the $\mathcal{I}(\ell, i)$ values by increasing distance k' from the diagonal, and stop once there are no more reachable speeds. That is, we start by computing all $\mathcal{I}(i, i)$, for increasing i. Observe that these values correspond to having $k' = 0$ outliers. Once we have all sets $\mathcal{I}(i-k', i)$ for some k', we continue with the $\mathcal{I}(i - (k' + 1), i)$ sets (see Fig. 2.7). Let k be the number of outliers in a maximum-length consistent subtrajectory, then all sets of speed intervals $\mathcal{I}(i - (k + 1), i)$ will be empty. Hence, the algorithm finishes after at most $k + 1$ "rounds". To compute

a single entry $\mathcal{I}(i - k', i)$ we have to propagate the speed intervals from at most k other entries (since all sets $\mathcal{I}(\ell, i)$ with $\ell > i$ are also empty). It follows that in total, this procedure takes $O(nk^2 \cdot P)$ time, where P is the time required to propagate all speed intervals in some set $\mathcal{I}(\ell', i)$ to $\mathcal{I}(\ell, j)$. Every set $\mathcal{I}(\ell, i)$ contains at most $\delta_{\max}$ intervals, which we keep in sorted order. Propagating a single interval takes constant time (see Lemma 2.4.2), and merging it with the intervals already in $\mathcal{I}(\ell, i)$ then takes $O(\log \delta_{\max})$ time.

Theorem 2.4.5. *Let T be a 1D trajectory with n probes. Under the acceleration-bounded model, the maximum length of a physically consistent subtrajectory of T can be computed in $O(nk^2\delta_{\max} \log \delta_{\max})$ time using $O(nk\delta_{\max})$ space, where k denotes the number of outliers and $\delta_{\max}$ the maximum fragmentation.*

▸ 2.4.4 Retrieving the physically consistent subtrajectory

The dynamic program computes the length ℓ^* of a maximum consistent subsequence. Generally, keeping track of the choices made in a dynamic program allows easy recovery of the actual answer, that is, the actual subsequence. However, we need slightly more, as we join overlapping intervals and thus no longer store which previous probes led to parts of that interval – generally there may not be only one probe for an interval.

We could opt for storing a minimum cover of the interval in a cell instead, which we can easily obtain while computing the union. However, this increases memory requirements. Alternatively, we can also use "backpropagation". That is, we extract S itself using the speed intervals in the sets $\mathcal{I}(\ell^*, i)$. We take an interval $I \in \mathcal{I}(\ell^*, i)$ and use an inverse propagation to find a probe p_h such that $\mathcal{I}(\ell^* - 1, h)$ has a nonempty interval of speeds at which the interval of $\mathcal{I}(\ell^*, i)$ is reachable. We repeat this backpropagation, until the start of the subsequence is reached.

To do this efficiently, we leverage that the intervals in $\mathcal{I}(\ell^* - 1, h)$ are sorted by the dynamic program already. Thus, we use backpropagation in $O(1)$ time by Lemma 2.4.2 to find the velocity interval I' at p_h. We then find whether one of the intervals in $\mathcal{I}(\ell^* - 1, h)$ intersects I' using binary search in $O(\log \delta_{\max})$ time. Thus, computing the subtrajectory after the dynamic program takes $O((n - k) \log \delta_{\max})$ time.

▸ 2.4.5 Bounding the maximum fragmentation

The running time of the dynamic program described in Section 2.4.1 depends on the maximum fragmentation $\delta_{\max}$, that is, the maximum number of intervals in

any set of velocities $\mathcal{I}(\ell, i)$. Recall that $\mathcal{I}(\ell, i)$ may contain more than one velocity interval if depending on which probe is contained earlier in the subsequence different restrictions on the speed at i, such as in the case of Fig. 2.1 (right). We argue in the following lemma that the fragmentation of a linear number of sets $\mathcal{I}(\ell, i)$ may even be $\Theta(n)$.

Lemma 2.4.6. *There is a 1D trajectory T with n probes such that $\Omega(n)$ sets of velocities $\mathcal{I}(\ell, i)$ have fragmentation $\Theta(n)$.*

Proof. We construct a trajectory $T = \langle p_1, \dots, p_{n/2}, q_1, \dots, q_{n/2} \rangle$ such that for parameters $v^- = 0$ and $a = a^+ = -a^- = 1$, we get $\Omega(n)$ speed intervals at each probe p_j, $j > n/2$.

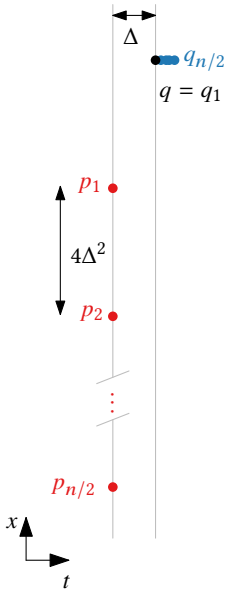


Figure 2.8: Instance with $\Theta(n)$ disjoint speed intervals at c.

Let $\Delta > 0$ be some real number. We place the probe locations at $p_i = (-4i \cdot \Delta^2, 0)$, for $i \in \{1, \dots, n/2\}$, and $q_j = (0, \Delta)$, for $j \in \{1, \dots, n/2\}$ (see Fig. 2.8). We can ensure that all probes have unique time stamps by offsetting them by some arbitrarily small time. This construction ensures that a consistent subtrajectory cannot use two probes p_i and p_j simultaneously. We now claim that every probe p_i together with probe $q = q_1$ generates a consistent subtrajectory $\langle p_i, q \rangle$ for which the possible speeds at q are

given by the interval $I_i = [v_i - \Delta, v_i + \Delta]$ with $v_i = 4i\Delta$. Observe that these intervals are all pairwise disjoint. Since the other probes q_j are arbitrarily close to q, the same argument shows that we get $\Omega(n)$ speed intervals at those probes.

Since $a = 1$ and the time between p_i and q is short, the velocity that the entity has at p_i must be similar to its velocity at q. If the speed at p_i differs too much from the velocity at q, then the entity cannot actually reach q: it will either travel too little or too far. Next, we formalize this argument.

To derive a contradiction, assume that there is a consistent subtrajectory in which an entity travels from p_j, with $j \neq i$, to q and arrives at q with speed $v \in I_i$. Since $v^- = 0$, the distance that any entity can and has to travel to go from p_j to q is exactly $4j\Delta^2$. The entity covers this in Δ time, and hence its average speed must be $4j\Delta$. Since $a = 1$, it then follows that at any time in the time interval $[0, \Delta]$ its speed lies in the range $[4j\Delta - \Delta, 4j\Delta + \Delta]$.

The entity achieves speed $v \in I_i = [v_i - \Delta, v_i + \Delta]$ at q. So, we have $4j\Delta - \Delta \leq v \leq v_i + \Delta$. Using that $v_i = 4i\Delta$ we get $j \leq i + \frac{1}{2}$. As i and j are natural numbers, we get $j \leq i$. Symmetrically, we have $v_i - \Delta \leq v \leq 4j\Delta + \Delta$, and get $i \leq j$. Combining these results gives $i = j$: a contradiction.

Note that in this construction all consistent subtrajectories have length two. We can easily achieve length $\ell > 2$ by prefixing the construction with a common trajectory of length $\ell - 2$; this prefix provides sufficient time between its last probe and the probes p_i, to allow the entity to achieve all speeds v_i at p_i. □

It is relatively easy to see that the fragmentation $\delta(\ell, i)$ is at most $O(2^i)$, since any fixed subsequence of $\langle \dots, p_i \rangle$ yields only a single interval (refer to Lemma 2.4.1). To realize such a large number of intervals, they have to be packed ever more closely to the minimum or maximum allowed speed threshold. It seems unlikely that this behavior will appear in realistic settings, and hence we expect that the fragmentation is much smaller in practice. Below, we hence describe an acceleration-bounded model which introduces some slack in the parameters a^- and a^+, which models real-world imprecision.

An acceleration model with slack In a real-world setting we can assume that there is some error in the parameters a^- and a^+ which bound the acceleration. To model this error we introduce a slack parameter $\varepsilon > 0$ for the acceleration bounds. Specifically, let $\Delta a = a^+ - a^-$ denote the difference between minimum and maximum acceleration. For our slacked bounds we add $\frac{\varepsilon}{2}\Delta a$ to a^+ and $-\frac{\varepsilon}{2}\Delta a$ to a^-. During the dynamic program, we first propagate intervals as usual, using the actual bounds a^- and a^+. Then we also propagate using the slacked bounds $a^+ + \frac{\varepsilon}{2}\Delta a$ and $a^- - \frac{\varepsilon}{2}\Delta a$. This is illustrated in Fig. 2.9: the green intervals are the result of standard propagation

and the red intervals are the result of slacked propagation. If two slacked intervals intersect, then we merge the corresponding standard intervals and use the merged interval for future propagation (in Fig. 2.9 the blue interval is the result of the merge).

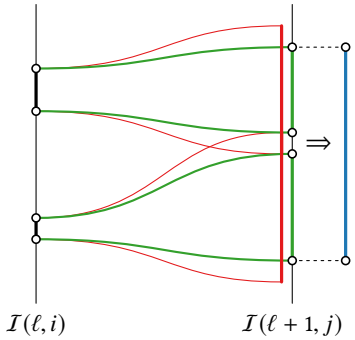


Figure 2.9: Propagation using the slacked model, from p_i to p_j. Green indicates standard propagation and red slacked propagation. The result of merging the intervals is indicated in blue.

In the following we give an upper bound on the size of any set of intervals $\mathcal{I}(\ell, i)$ as a function of ε. To do so, we estimate the number of disjoint intervals that can occur after propagation. First of all, note that at both the minimum and the maximum velocity, standard intervals can degenerate to a point. Slacked intervals, however, always have non-zero size. Consider now an interval $[\mathrm{v}, \mathrm{w}]$ which slack-propagates to a slacked interval $[\mathrm{v}_s, \mathrm{w}_s]$ of minimum size. This implies that in fact $\mathrm{v} = \mathrm{w}$, that is, the input interval degenerates to a point. We want to determine the minimum separation between v and any other input interval whose slacked propagation touches $[\mathrm{v}_s, \mathrm{w}_s]$. To this end, we compute the largest input velocity v_{hi} which slack-propagates to v_s. The separation is now given by $|\mathrm{v} - \mathrm{v}_{hi}|$. We can now compute a coarse upper bound for the number of intervals by dividing the complete input range $\Delta a \Delta t$ (see Section 2.4.2) by the separation:

$$\delta_{\text{prop}} = \frac{\Delta a \Delta t}{|\mathrm{v} - \mathrm{v}_{hi}|}$$

Solving for δ_{prop} results in

$$\delta_{\text{prop}} = \left\lfloor \frac{\sqrt{2}}{2\varepsilon\sqrt{\frac{1}{\varepsilon}+1}\left(\sqrt{2\sqrt{2}\sqrt{\frac{1}{\varepsilon}+1}-1}-1\right)} \right\rfloor = O(\varepsilon^{-1/4}),$$

which proves Theorem 2.4.7 below.

Theorem 2.4.7. *Let T be a 1D trajectory with n probes. Under the slacked acceleration-bounded model, the maximum fragmentation δ_{max} for any set of velocities $\mathcal{I}(\ell, i)$ is $O(\varepsilon^{-1/4})$.*

▸ 2.4.6 Extending to higher dimensions

The algorithm described above works for one-dimensional data. This may be realistic in some scenarios: for example, if we track contestants in a race along a predefined route, the known route defines an approximately one-dimensional space. However, in most cases, movement is in two or even three dimensions. There are various ways of generalizing the acceleration-bounded model.

There are two standard 2D "interpretations" of our algorithm: either we use the Euclidean distance between the probes, or we consider the Euclidean length of the path through all intermediate probes. In our view, the former is more suitable as we aim to remove outliers which could greatly affect distances in the latter.

Yet, assuming a linear motion between two probes is unrealistic as well. Thus, we use the Euclidean distance between probes only as a lower bound; the upper bound is the Euclidean distance multiplied by a constant λ. Note that an upper bound can also be derived from the current speed and acceleration bounds, but we use our simpler model in the experiments below. To propagate a velocity interval, we use the distance lower bound to determine the minimum velocity at the next probe, and the upper bound for the maximum velocity.

Of course, the models above assume that the tracked object may turn arbitrarily fast. Effectively, this means that positive or negative velocity becomes meaningless as we can instantaneously rotate from one to the other. We thus set the minimal velocity to zero. However, the direction of movement cannot be changed arbitrarily fast in reality, especially at higher speeds. Though we can easily define various physics models to address this issue, this would require more complex algorithms: we need to know more than just speed for the propagation and thus must generalize from intervals to higher-dimensional regions.

2.5 Experiments

We introduced various algorithms for computing maximum consistent subsequences of a trajectory, according to different physics models, specifically a speed-bounded and an acceleration-bounded model. The algorithms for the former are simpler and faster than for the latter. However, the acceleration-bounded model is more accurate. Through a series of experiments, we investigate the quality of our algorithms and the trade-off between them.

2.5.1 Algorithms

We use the following seven algorithms in our experiments. The first two refer to our optimal output-senstive algorithms described above, their running time depending on the number of outliers. Additionally, we use three comparison algorithms to investigate the quality of our methods with respect to simpler algorithms. These algorithms are two variants of an incremental greedy algorithm (under both physics models) and a local greedy method (under the speed-bounded model). We implemented all algorithms in C++; these implementations are open source and available as part of the MoveTK library.

[OSB] Optimal Speed-Bounded This algorithm implements the method of Section 2.2, under the speed-bounded model.

[OAB] Optimal Acceleration-Bounded This algorithm implements the method of Section 2.4. We use the 2D generalization, using $\lambda = 1.5$: the upper bound on the traveled distance is 1.5 times the Euclidean distance between two probes.

[GSB/GAB] Greedy Speed/Acceleration-Bounded We greedily build a consistent subsequence by testing whether the next considered probe is consistent with the last probe in the current subsequence under the speed-bounded model (GSB) or acceleration-bounded model (GAB). For GAB, we use the propagation technique of OAB to maintain an interval of speeds – the next probe is consistent if the interval after propagation is nonempty. These methods run in $O(n)$ time.

[SGSB/SGAB] Smart Greedy Speed/Acceleration-Bounded We keep track of multiple subsequences simultaneously. We append the next probe to each subsequence ending in a consistent probe ; if no such subsequence exists, the probe starts a new subsequence. The longest subsequence is returned. These methods run in $O(n^2)$ time.

Table 2.1: Summary of the complexities and speeds of trajectories per data set. The final columns list the default model parameters used throughout the experiment. v^+ is given in km/h, a^- and a^+ in m/s^2.

	trajectories	complexity		speed (km/h)		model parameters		
		mean	maximum	stddev	mean	stddev	v^+	a^-	a^+
MB	1 214	3 377.1	22 426	2 643.4	18.8	10.9	35.0	−3.24	1.62
HR	5 000	424.9	8 925	545.6	62.3	43.0	125.0	−10.00	10.00
LA	78 658	304.4	38 719	1 082.0	55.8	1 557.7	129.6	−10.00	10.00

[LGSB] Local Greedy Speed-Bounded Zheng [139] points us to another method that uses a speed bound for outlier detection. To the best of our knowledge this is the only other such method described in the literature. However, neither Zheng's survey nor the references therein give a detailed description of this heuristic method. We hence compare against our interpretation of the sketch provided by Zheng [139]. We construct a graph with a vertex per probe . Two vertices are connected if their probes are successive in the original trajectory and they are consistent according to the speed bound. A probe is added to the output, if and only if its vertex is in a connected component of a user-specified size; we set this value to 3 in our experiments. Note that this local heuristic does not guarantee that the complete output is consistent according to the speed bound. This method runs in $O(n)$ time.

▸ 2.5.2 Data sets

We use the data sets **MB**, **HR**, and **LA**. See Appendix A for details of these data sets.

All trajectories in the data sets are filtered to have at least 10 probes. General statistics of these data sets are provided in Table 2.1, along with our parameter settings per data set, which are based on the nature and location of the general data set; note that v^- is always set to 0 to allow the tracked object to remain stationary.

▸ 2.5.3 Comparing algorithms and models

In our analysis of the results, we look primarily at relative lengths, that is, the ratio of the number of probes with respect to the input size. Thus, a result that filters k outliers and keeps $n-k$ probes has a relative length of $\frac{n-k}{n} \in [0,1]$. In the remainder, we simply use length to refer to relative length. We start, however, with a brief consideration of efficiency.

Efficiency Table 2.2 provides performance statistics per algorithm and data set in terms of running time, as performed on a HP Elitedesk 800 g2 TWR (Intel Core i5-6500 CPU at 3.20GHz; 16 GB of RAM; 64-bit Windows 10 Enterprise). Overall, the trend between the algorithms per data set is roughly the same. We see differences between data sets – specifically MB with respect to LA and HR – which are simply a result of the increased trajectory complexity within the MB data set. We see that our OSB is competitive with GSB and even considerably faster than SGSB. As we may expect from the theoretical analysis, OAB is very slow in comparison to the other algorithms, yet the greedy alternatives are comparatively fast.

The main questions to investigate are thus two-fold: (1) is the speed-bounded model able to achieve reasonable results, compared to the acceleration-bounded model? (2) how do the faster greedy approaches compare in terms of quality with respect to the optimal algorithms. We first investigate the latter question before turning to the former.

Speed-bounded model We have three algorithms that strictly adhere to the speed-bounded model: OSB, GSB and SGSB (see left two columns of Fig. 2.10). As OSB computes optimal results, GSB and SGSB cannot result in longer subsequences. For the MB data set, we observe that GSB and SGSB perform very similarly in terms of the number of outliers detected. For the HR and LA data sets we see larger differences, especially for GSB. Table 2.3 shows the ratio between OSB and GSB/SGSB according to different brackets of OSB. These numbers indicate that a vast majority of trajectories has less than 10% outliers, and that in such cases the results are on average not much different. The more outliers are present, the more pronounced the difference between our optimal result and the greedy results becomes.

OSB is thus more reliable, as it gives optimal results. When there are few outliers, this algorithm is close to linear and thus we may expect less of a performance loss compared to the simpler methods. Indeed, we see that in terms of running time, OSB (0.48 ms on average per trajectory) performs similarly as the GSB (0.24 ms) and is actually faster than SGSB (5.35 ms). When there are many outliers, the extra time spent may be well worth the effort to obtain the maximum consistent subsequence.

Table 2.2: Mean, 99 percentile, and maximum running time in milliseconds (ms), unless indicated otherwise. Running times are shown per data set, and over all data sets. Note that the imbalance in data set size skews the mean over all data sets strongly to the mean of the LA data set.

	MB			HR			LA			All data sets		
	mean	99%	max	mean	99%	max	mean	99%	max	mean	99%	max
OSB	5.32	31.25	203.12	1.01	15.62	62.50	0.37	15.62	359.38	0.48	15.62	359.38
GSB	2.37	15.62	15.62	0.32	15.62	15.62	0.20	15.62	31.25	0.24	15.62	31.25
SGSB	210.26	1 812.50	7 750.00	4.15	78.12	468.75	2.27	15.62	8 593.75	5.35	78.12	8 593.75
LGSB	2.59	15.62	31.25	0.30	15.62	15.62	0.21	15.62	31.25	0.25	15.62	31.25
OAB	7 194.19	89.1**s**	1 074.6**s**	71.03	1 640.62	14.7**s**	127.04	171.88	1 754.6**s**	224.83	1 156.25	1 754.6**s**
GAB	4.22	15.62	31.25	0.45	15.62	15.62	0.35	15.62	46.88	0.41	15.62	46.88
SGAB	95.37	921.88	2 656.25	2.35	31.25	234.38	1.47	15.62	4 843.75	2.86	46.88	4 843.75

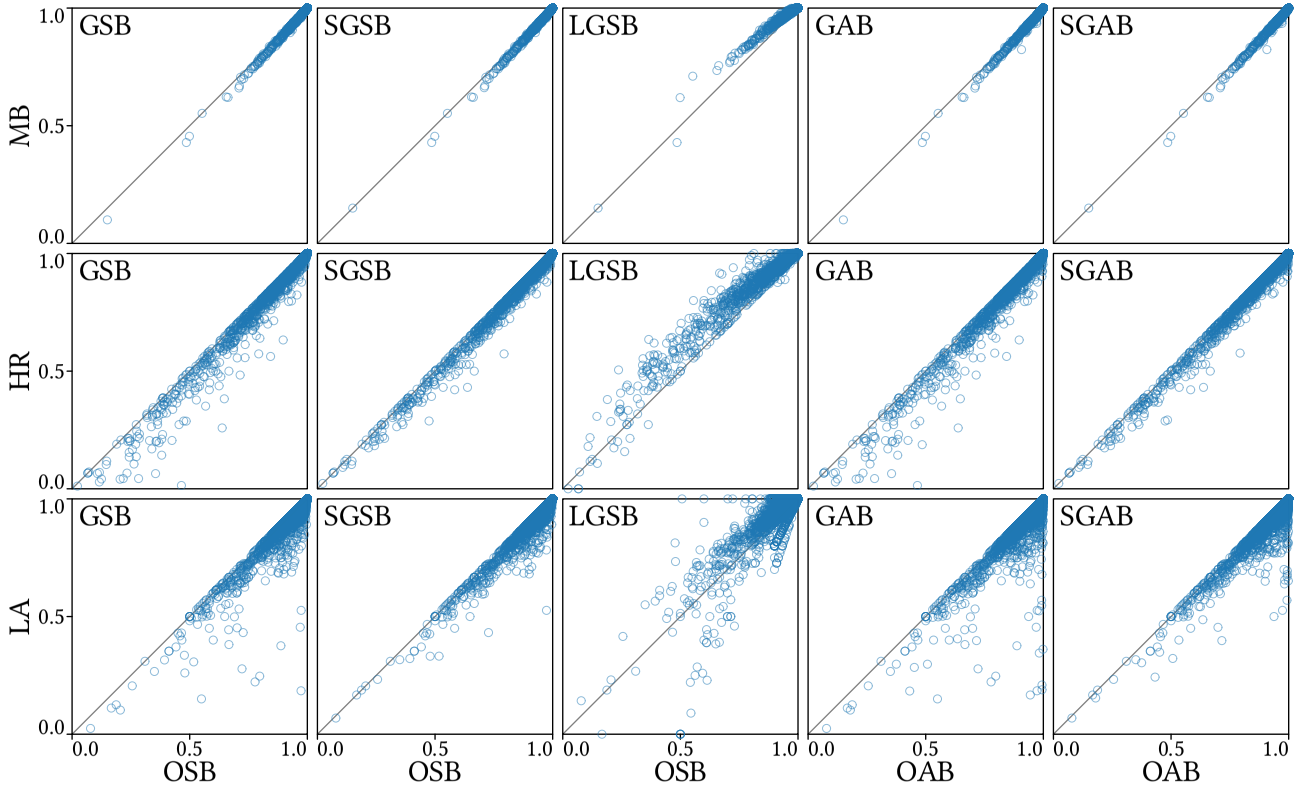


Figure 2.10: Comparing the various algorithms. Each axis represents the (relative) length. Top row: MB data; middle row: HR data; bottom row: LA data. First three columns: comparison of OSB with GSB, SGSB and LGSB; last two columns: comparison of OAB with GAB and SGAB.

Table 2.3: Mean and standard deviation of the ratio between greedy strategies and optimal strategies, split by bins of the optimal length ("length" row). The "size" row indicates the percentage of trajectories in the corresponding length bin. in GSB, SGSB and LGSB are compared to OSB; GAB and SGAB to OAB. Table is continued on the next page.

			MB	
Length	**0.0 - 0.6**	**0.6 - 0.8**	**0.8 - 0.9**	**0.9 - 1.0**
Size	0.07%	0.20%	0.57%	99.17%
GSB	0.87 ± 0.14	0.97 ± 0.01	0.98 ± 0.01	1.00 ± 0.01
SGSB	0.95 ± 0.06	0.97 ± 0.01	0.98 ± 0.01	1.00 ± 0.01
LGSB	1.10 ± 0.20	1.08 ± 0.02	1.05 ± 0.01	1.01 ± 0.01
Size	0.07%	0.21%	0.70%	99.02%
GAB	0.98 ± 0.06	0.97 ± 0.01	0.98 ± 0.01	1.00 ± 0.01
SGAB	1.10 ± 0.27	0.97 ± 0.01	0.98 ± 0.01	1.00 ± 0.01
			HR	
Length	**0.0 - 0.6**	**0.6 - 0.8**	**0.8 - 0.9**	**0.9 - 1.0**
Size	3.14%	4.58%	5.96%	86.32%
GSB	0.83 ± 0.20	0.95 ± 0.07	0.98 ± 0.03	1.00 ± 0.01
SGSB	0.93 ± 0.07	0.97 ± 0.04	0.99 ± 0.02	1.00 ± 0.01
LGSB	1.22 ± 0.28	1.11 ± 0.07	1.05 ± 0.03	1.00 ± 0.01
Size	3.14%	4.64%	5.94%	86.32%
GAB	0.83 ± 0.21	0.95 ± 0.07	0.98 ± 0.03	1.00 ± 0.01
SGAB	0.93 ± 0.08	0.97 ± 0.04	0.98 ± 0.02	1.00 ± 0.01

Table 2.3: Mean and standard deviation of the ratio between greedy strategies and optimal strategies, split by bins of the optimal length ("length" row). The "size" row indicates the percentage of trajectories in the corresponding length bin. in GSB, SGSB and LGSB are compared to OSB; GAB and SGAB to OAB.

	LA			
Length	**0.0 - 0.6**	**0.6 - 0.8**	**0.8 - 0.9**	**0.9 - 1.0**
Size	0.33%	2.06%	7.00%	90.61%
GSB	0.87 ± 0.17	0.93 ± 0.11	0.97 ± 0.05	1.00 ± 0.01
SGSB	0.94 ± 0.08	0.96 ± 0.05	0.98 ± 0.03	1.00 ± 0.01
LGSB	1.05 ± 0.48	1.04 ± 0.16	1.05 ± 0.05	1.00 ± 0.01
Size	0.33%	2.06%	7.33%	90.28%
GAB	0.87 ± 0.17	0.93 ± 0.11	0.97 ± 0.04	1.00 ± 0.01
SGAB	0.93 ± 0.09	0.96 ± 0.06	0.98 ± 0.03	1.00 ± 0.01

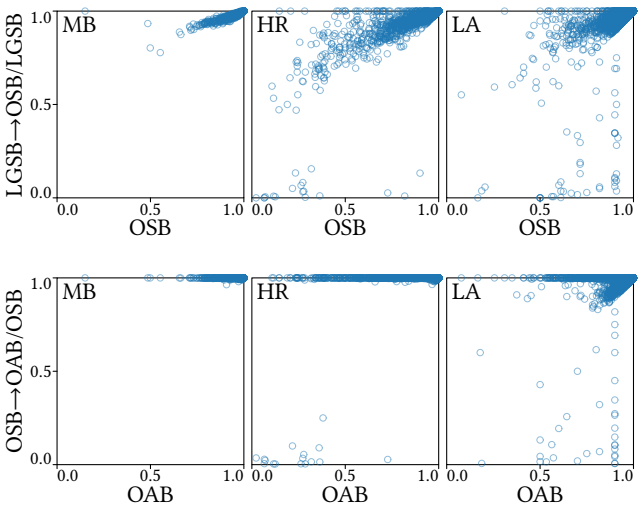


Figure 2.11: Postprocessing to ensure a stricter physics model. Left column: MB data; middle column: HR data; right column: LA data. Top row: comparison of LGSB⟶OSB with OSB; bottom row: comparison of OSB⟶OAB with OAB.

Acceleration-bounded model Referring to Fig. 2.10 and Table 2.3, we observe the same patterns between OAB and GAB/SGAB as above for the speed-bounded variants, but the differences are more pronounced. However, it must be noted that the computation times behave much differently. Although the number of intervals in a single cell never exceeds 2 for almost all trajectories (with a maximum of 4), the computation time of OAB (224.8 ms on average per trajectory) is significantly higher than GAB (0.41 ms) and SGSB (2.86 ms). Thus, OAB seems practical mostly for cases where processing speed is not a primary concern: for example, because much longer offline computations are expected afterwards, or because the trajectory lengths are limited.

Local strategy The LGSB method can also be compared to OSB. However, because this method does not ensure that the entire subsequence adheres to the physics model, it may be the case that LGSB yields a longer sequence than OSB. This is quite structurally the case (see third column in Fig. 2.10), with more pronounced effects for a large number of outliers (see Table 2.3, LGSB rows). This indicates that the local strategy for determining outliers is not quite suitable for capturing the actual constraints of the physics model.

We further investigate the local method by postprocessing the results of LGSB by OSB (LGSB⟶OSB). That is, we find the longest consistent subsequence of the LGSB result. If LGSB would work perfectly, no outliers are filtered in this postprocessing step. The more outliers are found in the LGSB result, the more violations of the physics model the LGSB result exhibits. The top row of Fig. 2.11 shows the results; note that the vertical axis shows (relative) length of the final result with respect to the length without postprocessing rather than (relative) length with respect to the input. We see again that the results depend on the number of outliers in the trajectory, but overall the difference may be quite pronounced: LGSB⟶OSB on average has 8.75% less probes than LGSB for cases with OSB length less than 0.9. The data set also has an effect: MB has less variance than the other two data sets.

Comparing models Since any acceleration-bounded path in our setting is also a speed-bounded path, OSB cannot detect more outliers than OAB. That is, OSB results can be interpreted in the acceleration-bounded model and we can investigate how well the model inherently meets the acceleration bound. We follow the same approach in comparing LGSB to OSB above, postprocessing OSB results by OAB (OSB⟶OAB) to determine how many outliers the OSB result still includes according to the stricter model.

The bottom row of Fig. 2.11 shows the results. We clearly see that that only few probes are filtered in the OAB postprocessing step for all three data sets. This pattern is strongest in MB (0.09% classified as outliers on average) and HR (0.04% on average), even for more noisy trajectories. For the LA data set, slightly more probes are filtered (1.74% on average), but interestingly, this seems mostly the case for the less noisy trajectories. These averages are based on the cases with OAB length less than 0.9.

We may conclude that generally the speed-bounded model is capable of getting quite realistic results even for the acceleration-bounded case, while avoiding the computational complexity. It is interesting that there seems to be slightly different behaviors between the two vehicle data sets: this raises the question whether differences in traffic and driving behavior make acceleration more important in certain environments than in others.

▸ 2.5.4 Sensitivity of model parameters

The physics models have a few parameters to capture what is considered feasible movement through space and time. Here, we look at how sensitive the results are to changing the parameter values. Following our observations from the previous

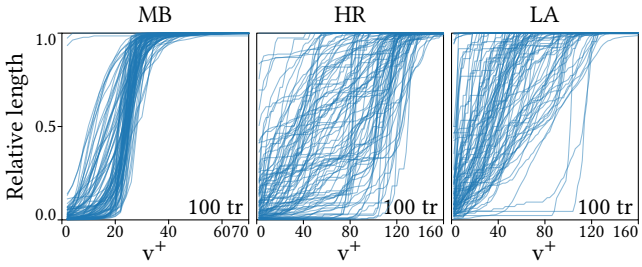


Figure 2.12: Profile of the speed bound: length of OSB as a function of v^+, for 100 random trajectories for each data set.

Table 2.4: Sensitivity σ of the speed bound v^+.

data set	mean	stddev	min	99%	max
MB	0.093	± 0.049	0.013	0.243	0.368
HR	0.059	± 0.042	0	0.244	0.418
LA	0.047	± 0.033	0	0.180	0.458

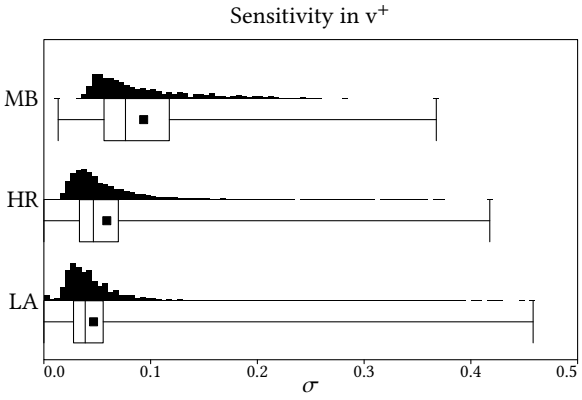


Figure 2.13: Sensitivity σ of the speed bound v^+ per data set.

section, we focus on the speed-bounded model which effectively has one parameter: the maximum speed v^+, but we also briefly investigate the effect of the detour factor as well as the acceleration bound in OAB.

Procedure Our analysis for each parameter follows the same procedure: we vary the parameter systematically from very restrictive values to very generous values, running the optimal algorithm for the model under consideration on all data. We then consider how the length of the result varies with this parameter.

To allow for summarizing the results, we operationalize the sensitivity σ for a single trajectory as the maximum of the difference between relative length and the difference between two parameter values, over all (consecutive) pairs of parameter values. We refer to the the two parameters that result in the maximum the *sensitive range* ρ of that trajectory; we use the mean value of the range to compute summary statistics. The unit of σ is thus the inverse of the unit of the parameter, but we generally omit this indication. Intuitively, the sensitivity is the "slope" when plotting the relative length as a function of the parameter, which we refer to as a *profile*. We illustrate these functions for each case using a selection of the trajectories for each data set, consider summary statistics over all trajectories, and investigate the relation between the sensitivity and the sensitivity range.

Note that our choice of step size in varying the parameter inherently limits the maximum sensitivity that can be obtained to the reciprocal of the step size. For example, steps of 2 km/h in varying the speed bound v^+, limits the sensitivity to 0.5, which would indicate jumping from length 0 to 1 between two values of v^+. In degenerate, constructed inputs this can indeed be realized – in fact, any arbitrarily large sensitivity can achieved in theory. Consider a hypothetical trajectory of n probes along a straight line, sampled every second, with a distance between consecutive probes a distance c. The length of the optimal result for any $v^+ < c$ is then $1/n$, as no pair is consistent. However, the length for $v^+ \geq c$ is 1. Thus, for $v^+ = c$ and $v^+ = c - \varepsilon$ we obtain a sensitivity of $(1 - 1/n)/\varepsilon$. For ε approaching zero, this thus tends to infinity. Thus, we focus on the practical slope of these curves, using some reasonable sampling of the domain of the parameter.

Speed bound We run our OSB algorithm, using a speed bound v^+ from 2 km/h to 70 km/h (MB), from 2 km/h to 160 km/h (HR and LA), in steps of 2 km/h. Fig. 2.12 provides a random sample of the resulting profiles. As we can see, many trajectories follow the roughly the same pattern of a few steep increases at different speed bounds. We attribute this to different behavior of the moving entity. For MB, this behavior is fairly consistent, with a high sensitivity around 21.5 km/h (average sen-

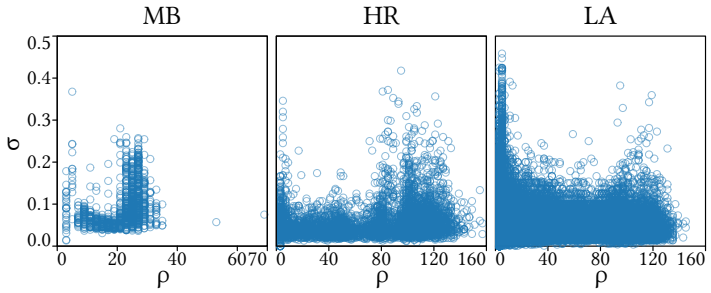


Figure 2.14: Scatterplot relating the sensitivity σ of v^+ to (the mean of) the sensitivity range ρ. Each circle represents a single trajectory.

sitivity range). For the other data sets, this is less clear, likely reflecting different driving behavior due to local speed limits, which varies between trajectories but also within a single trajectory.

Table 2.4 and Fig. 2.13 show summary statistics of the sensitivity for the three data sets. We see that the sensitivity can be quite high in extreme cases: changing the parameter by 1 km/h may change the relative length by almost 0.46. On average, the sensitivity is considerably lower. However, these results still show that careful selection of the model parameters is important: too low values result in probes being identified as outliers unjustly, but setting them too high might leave too many outliers undetected.

With Fig. 2.14, we look at the relation between the sensitivity and the sensitivity range. We roughly see the same pattern for each of the data sets: a number of trajectories have their relatively large sensitivity at low speeds, followed by another peak at higher speeds. Potentially, this separates the trajectories into different cases of actual behavior: for example, cars drive at different speeds in residential areas, provincial roads and highways – if a trajectory falls mostly within one of the categories, it is reasonable to expect the largest sensitivity to occur at that speed. Under this hypothesis, we see that we used quite a reasonable bounds on the maximum speed, that is, values slightly higher than the sensitivity range for majority of the trajectories.

Acceleration bound The acceleration-bounded model has, as the name suggests, parameters controlling the allowed acceleration and deceleration, a^+ and a^- respectively. Due to the high computational cost of OAB, we restrict our attention to only six values combinations of a^+ and a^- per data set. The parameters we selected for

defaults reflect fairly extreme capabilities: limits of racing cars (HR and LA) and estimates of well-trained cyclists (MB). To investigate the effect of these parameters, we thus reduce these parameter values, to reflect settings of "normal" and "slow" behavior in terms of acceleration and deceleration. Specifically, we test the following six combinations for each data set: $a^+ \in \{2, 4, 10\}$ and $a^- \in \{-2, -10\}$ (HR and LA); $a^+ \in \{0.8, 1.2, 1.62\}$ and $a^- \in \{-2, -3.24\}$ (MB). Each of these values is expressed in $\mathrm{m/s^2}$.

We can now study sensitivity of the one parameter by fixing the other parameter to each of its values. A sample of the resulting profiles are shown in Fig. 2.15 and Fig. 2.16. We immediately see that there is very little effect of the acceleration or deceleration bound. As these are not a random sample, but actually the profiles with highest sensitivity, this tells us that these parameters are of little influence.

The summary statistics over all trajectories further confirm this, as shown in Fig. 2.17. Considering our chosen set of parameters, the sensitivity in a^+ can be at most 0.5 (HR and LA) or 1.67 (MB); for a^- these maxima are 0.125 (HR and LA) and 0.81 (MB). What we observe, however, is that the actual sensitivity is significantly lower – also foregoing the need for further refine the tested parameter values. The strongest sensitivity observed is 0.066 for a^+ and 0.01 for a^-. This is, however, an "extreme" with medians and averages laying much closer to zero.

One observation to be made is that the sensitivity for the maximum acceleration in the MB seems to be slightly higher, though still much lower. This is possibly caused by the nature of the data: a mountain biker may accelerate and decelerate more strongly, compared to regular traffic.

In Fig. 2.18 and Fig. 2.19, we show histograms for the sensitivity, split by the sensitivity range. In these charts, we omit all trajectories which have sensitivity zero – the number of remaining trajectories is indicated per data set. Notably, we see that MB has relatively few trajectories with zero sensitivity, whereas for the other data sets this is the majority of trajectories. Again, we attribute this to the different nature of mountain biking.

In light of the little dependence on the acceleration bounds, we assume that the speed bounds used by the acceleration model, in terms of sensitivity, behave similarly as for the speed-bounded model. More importantly, these results further support our conclusion from Section 2.5.3: the speed-bounded model provides realistic results even for the acceleration-bounded model.

Detour factor The OAB algorithm uses a detour factor λ, to determine how much distance can at most be traveled between two probes, which is λ times the Euclidean distance. In other experiments, this is fixed to 1.5, but here we investigate how much

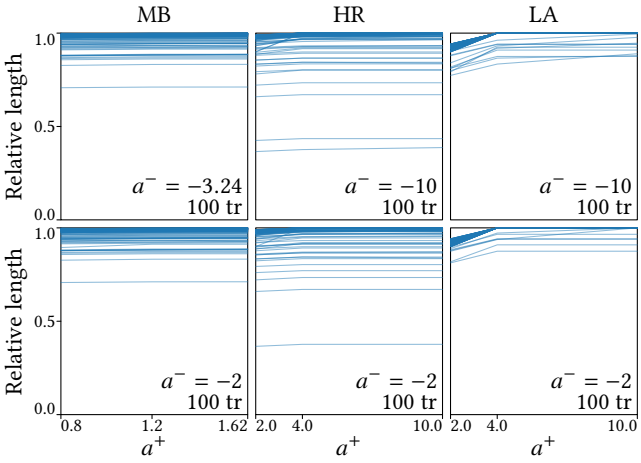


Figure 2.15: Profiles of the acceleration bound: length of OAB as a function of a^+, for the 100 most sensitive trajectories for each data set.

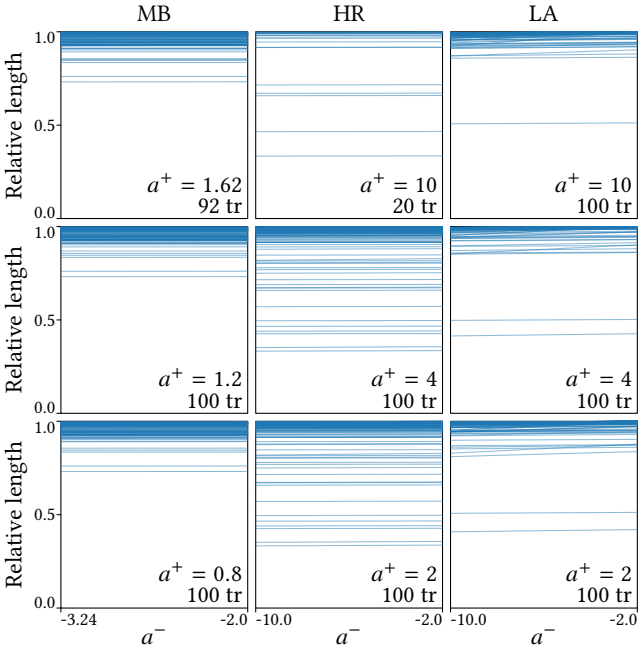


Figure 2.16: Profiles of the deceleration bound: length of OAB as a function of a^-, for the 100 most sensitive trajectories for each data set.

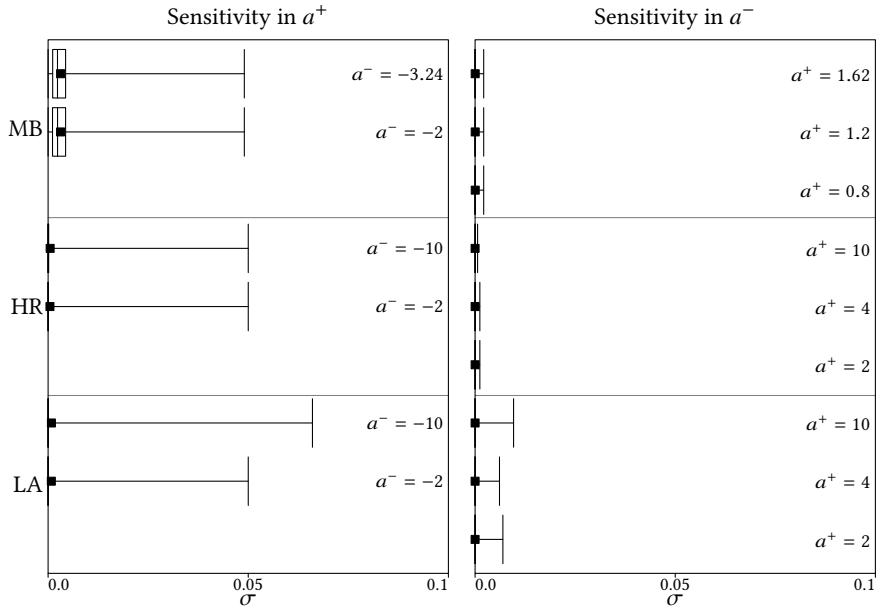


Figure 2.17: Sensitivity of a^+ (left) and a^- (right) per data set and value of the other parameter. Note that the technical maximum sensitivity would be significantly higher, but this does not occur – the horizontal scale has been adjusted.

this parameter may influence the results. We run the OAB algorithm using λ from 1 to 2, with increments of 0.1.

Refer to Figures 2.20 and 2.21. We observe that detour factor λ has very little influence in general, with most trajectories not being influenced by λ at all: 52 out of 1 214 for MB, 4 049 out of 5 000 for HR and 74 500 out of 78 658 for LA. The detour factor is likely to help in cases where turns are made at relatively high speed: the Euclidean distance might be too short to slow down and reach the next probe at the right time – but adding some slack gives enough space to travel between two somewhat close points at high speed. Thus, this factor can be expected to be of less influence for trajectories with high sampling frequency or without turns are relatively high speed. This may explain why the mountain-bike data set exhibits more sensitivity than the other two vehicle data sets.

Fig. 2.22 relates the sensitivity to the sensitivity range. We observe that the highest sensitivity is found in the sensitivity range [1, 1.1], and generally a trend of higher sensitivity at lower values of λ. This supports our suggestion above as to the

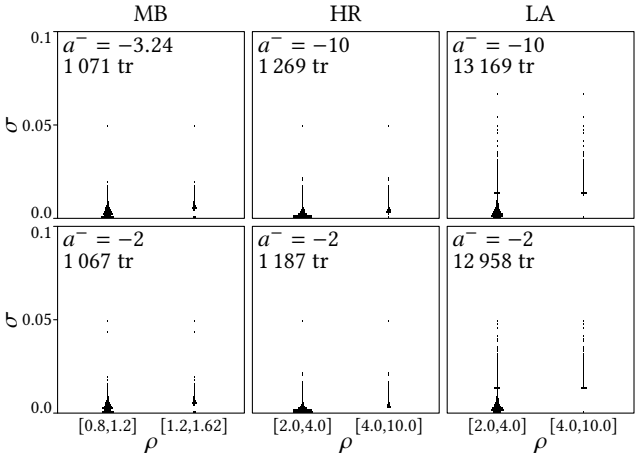


Figure 2.18: Histogram relating sensitivity of a^+ and the sensitivity range. Trajectories with sensitivity 0 have been omitted. Note that the technical maximum sensitivity would be significantly higher, but this does not occur – the horizontal scale has been adjusted.

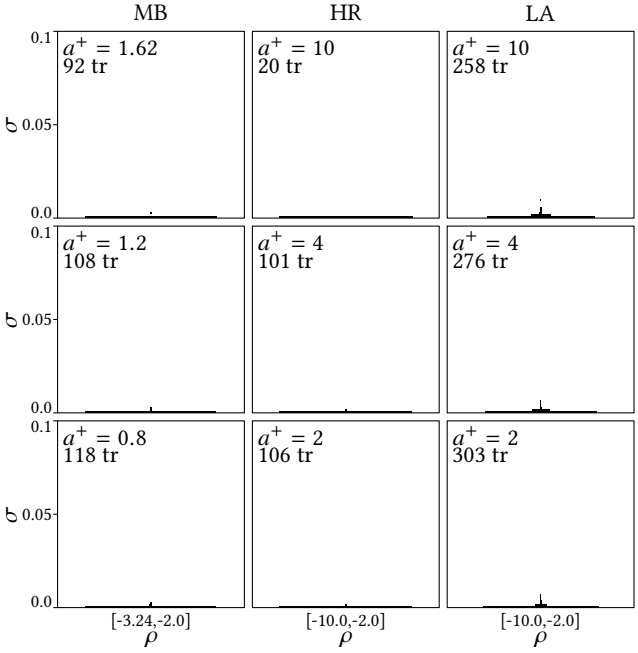


Figure 2.19: Histogram of sensitivity of a^-, of the single sensitivity range tested. Trajectories with sensitivity 0 have been omitted. Note that the technical maximum sensitivity would be significantly higher, but this does not occur – the horizontal scale has been adjusted.

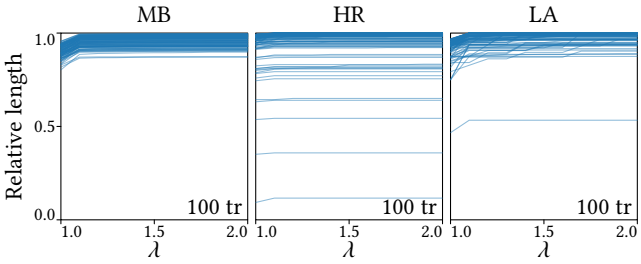


Figure 2.20: Profiles of the detour factor: length of OAB as a function of λ, for the 100 most sensitive trajectories for each data set.

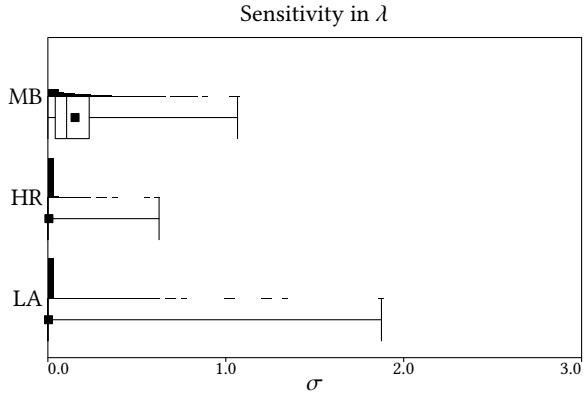


Figure 2.21: Sensitivity of λ per data set. Note that the technical maximum sensitivity would be 10, but this does not occur – the horizontal scale has been adjusted.

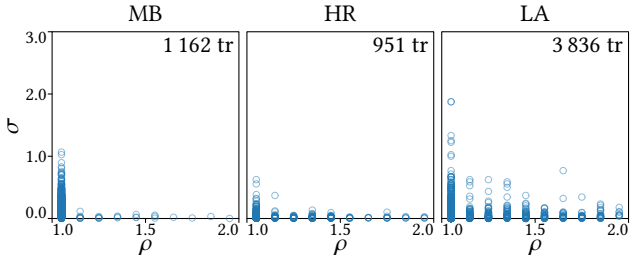


Figure 2.22: Scatterplot relating the sensitivity σ of λ to (the mean of) the sensitivity range ρ. Each circle represents a single trajectory. Trajectories with sensitivity 0 have been omitted. Note that the technical maximum sensitivity would be 10, but this does not occur – the vertical scale has been adjusted.

cause of the low sensitivity.

Most of our trajectories have relatively high sampling frequency and as such the sensitivity is low. The question is how these observations generalize to low sampling frequencies. This will likely depend strongly on the object being tracked. If it travels frequently at nearly maximum speed, sensitivity may be high. However, if the general speed is significantly lower, the admitted variation in the reconstructed speed may already be sufficient to avoid sensitivity in the detour factor.

▸ 2.6 Discussion

Results Our results indicate that our optimal algorithms outperform simple greedy strategies, either in quality of the results, running time, or both. Noise levels and other characteristics do influence these results, and our methods are particularly effective for dealing with large amounts of noise. The example in Fig. 2.23 (top) illustrates a case where the OAB algorithm computes a longer sequence, compared to SGAB: the cause is that a few erroneous probes lead this greedy algorithm to make a sequence that prevents it from selecting many probes later.

Furthermore, the results suggest that the quality difference between models with only a speed bound and acceleration-bounded models is small. This must be considered carefully though, as there is an effect of social or geographic environment. Fig. 2.23 (bottom) shows a case where the OSB algorithm detects fewer outliers, though the difference is only minor. Contrasting the previous comparison, this implies that OAB performed better than OSB: OSB fails to capture the outlier that is not physically realizable in the stronger acceleration-bounded model. That is, there is not enough time to realistically decelerate and accelerate to capture the full near-stationary probes.

The selection of parameters influences the results, but this is mostly the case for the maximum speed. Acceleration and detour factor for our OAB algorithm tend to have minimal effect on the number of outliers detected, though we observed variation between the types of moving entities. Fig. 2.24 shows a sequence of results for different speed bounds, for a trajectory from each data set. Increasing the speed bound leads to fewer outliers – but possibly less realistic behavior, if the bound is set too high. The effect of lowering the speed bound is that corners tend to be cut by marking outliers, to lower the traveled distance to one that is achievable within the speed bound.

Context By design, we do not consider the use of other data, such as a road network that a vehicle is driving on. However, such data opens up various potential

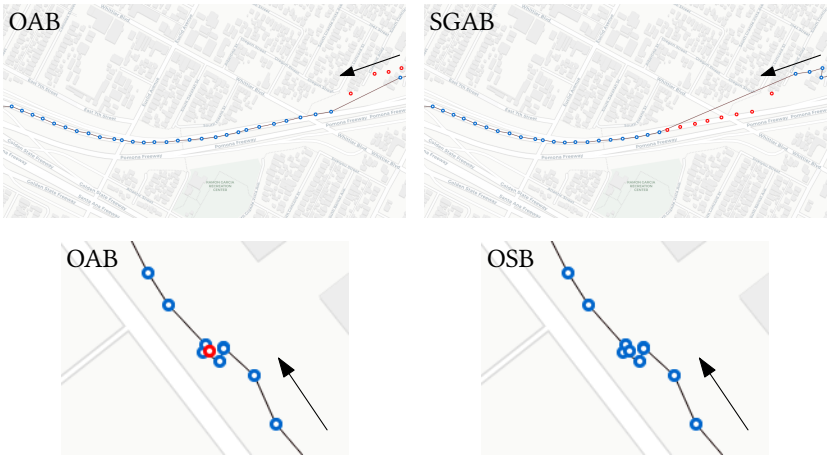


Figure 2.23: Example trajectory where OAB differs from SGAB (top) and OSB (bottom). Arrows indicate the direction of travel, blue markers are probes that are part of the consistent subsequence computed, and red markers indicate the corresponding outliers. Both trajectories are from the LA data set. Base maps ©OpenStreetMap.

avenues for further research. For example, given a road network, we may be able to more accurately assess the travel distance or limit it to a few likely candidates, rather than using the Euclidean distance. For OSB and OAB, this is straightforwardly included into the algorithm. For our faster algorithm under the speed-bound model, however, this is not quite the case, as the AWVD is no longer directly applicable, but there may be potential to generalize the approach.

Beyond assessing distances more accurately, additional data could also be used to define more accurate physics models. Our current models are fairly simple, and use only few parameters to define global thresholds on the maximum speed and acceleration. However, such thresholds may actually depend on the environment. For example, expected maximum speed for driving in a car is different on the highway than it is in an urban environment. Similarly, cycling uphill or downhill affects maximum speed. Ideally, physics models and, by extension, outlier-detection algorithms should accommodate for such variations, as this allows for more efficacious processing of heterogeneous trajectories that travel through different environments.

Including contextual factors will make the models more accurate and realistic, but a crisp decision boundary (movement is or is not physically possible) may no longer exist. Instead, we may want to define that a car can violate speed limits, but the severity and duration affect how likely the behavior is. Future work could

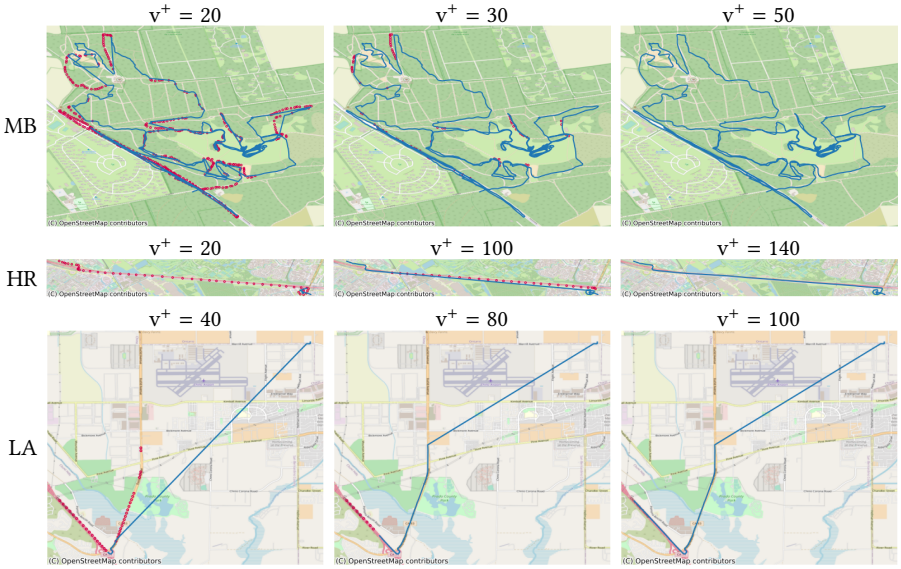


Figure 2.24: Example results, using different speed bounds v^+. One trajectory for each data set is shown. Blue line represents the resulting trajectory, with red dots marking the outliers. Base maps © OpenStreetMap.

explore "behavioral models" that describe expected movement more closely, including context, and are more robust by allowing deviations from the model, thereby reducing parameter sensitivity.

Enhancing other techniques There are many other forms of trajectory processing and analysis techniques, such as clustering, map matching, and segmentation. Such techniques may be complemented or enhanced by applying physics models to define possible or realistic behavior. For example, a map-matching algorithm could include considerations of whether its result is physically realizable, or clustering may be done based on what physical behavior would be necessary to realize certain trajectories. We leave exploring such complementarity of techniques to future work, but our results presented here provide a framework and methods that may be integrated into such enhanced techniques.

Chapter 3

Simplification

3.1 Introduction

To reduce storage and computation times, trajectories can be simplified as a *preprocessing task*. As discussed in Chapter 1, we can treat a trajectory as a polygonal curve if we ignore its timestamps. This allows the use of the large body of research already done in simplification of polygonal curves. The algorithmic questions we address in this chapter are all based on this abstraction of trajectories. As such, throughout this chapter we will be using the term “curve", as it is more general than “trajectory". To further generalize the terminology, in this chapter we will not refer to the probes that make up a trajectory as such, but we will only refer to them as vertices of a curve.

Curve simplification is a long-studied problem in computational geometry, geographical information science (GIS), and automated cartography. It also has applications in related disciplines such as graphics. Given a polygonal curve P with n vertices, the goal is to find another polygonal curve P' with a smaller number of vertices such that P' is sufficiently similar to P. Unless specified otherwise, in this chapter we will assume P is two-dimensional. Well-known methods proposed for this problem include a simple heuristic scheme by Douglas and Peucker [49], and a more involved classical algorithm by Imai and Iri [73]; both are frequently implemented and cited. Since then, numerous further results on curve simplification, often in specific settings or under additional constraints, have been obtained [6, 9, 21, 25, 36, 39, 42, 63, 72].

Despite its popularity, the Douglas-Peucker algorithm comes with no provable

quality guarantees. The method by Imai and Iri, though slower, was introduced as an alternative which does supply guarantees: it finds an optimal shortest path in a graph in which potential shortcuts are marked as either *valid* or *invalid*, based on their distance to the corresponding sections of the input curve. However, Agarwal et al. [9] note that the Imai-Iri algorithm does not actually globally optimize any distance measure between the original curve P and the simplification P'. This work initiated a more formal study of curve simplification; Van Kreveld et al. [89] systematically show that both Douglas-Peucker and Imai-Iri may indeed produce far-from-optimal results.

This raises a question of what it means for a simplification to be optimal. It can be seen as optimal to reduce the number of vertices as much as possible while having a bound on the curve distance between the original and the simplification. This is known as min-link simplification. Alternatively, we could set a bound on the number of vertices of the simplified curve and try to find a simplification with minimum distance, known as a min-δ simplification. Imai-Iri is a min-link algorithm, and this chapter also focusses on min-link simplification. For both min-link and min-δ simplification, which simplified curve is optimal depends on the distance measure that is used.

However, the difference in interpretation between Agarwal et al. and Imai and Iri lies not so much in the choice of distance measure, but rather what exactly the measure is applied to. In fact, the Imai-Iri algorithm is optimal in a *local* sense: it outputs a subsequence of the vertices of P such that the Hausdorff distance between each shortcut and *its corresponding section of the input* is bounded: each shortcut approximates the section of P between the vertices of the shortcut.

We underline this difference by using the term *global* simplification when a bound on a distance measure must be satisfied between P and P' (formal definition in Section 3.2.3), and *local* simplification when a bound on a distance measure must be satisfied between each edge of P' and its corresponding section of P. Clearly, a local simplification is also a global simplification, but the reverse is not necessarily true, see Figure 3.1.

Both local and global simplifications have their merits: one can imagine situations where it is important that each segment of a simplified curve is a good representation of the curve section it replaces (e.g. if the curve is a trajectory where we plan to later make use of the timestamps associated with the vertices), but in other applications (e.g., visualization) it is really the similarity of the overall result to the original that matters. Most existing work on curve simplification falls in the *local* category. We focus on curve simplification where the considered similarity constraint is a global distance measure.

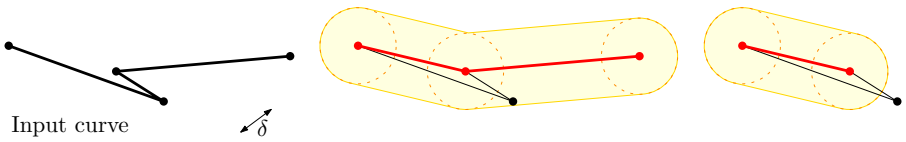


Figure 3.1: For a target distance δ, the red curve (middle) is a global simplification of the input curve (left), but it is not a local simplification, since the first shortcut does not closely represent its corresponding curve section (right). The example applies to both Hausdorff and Fréchet distance.

Our contribution is a systematic overview of different variants of the global curve simplification problem. We consider six different curve distance measures and three different restrictions on vertex placement, giving eighteen different variants in total. We have collected results in the literature for these variants, and supplemented these results with our own theoretical findings. See Table 3.1 for an overview of old and new results.

▸ 3.1.1 Existing work on global curve simplification

Surprisingly, only a few results on simplification under global distance measures are known [9, 24, 29, 89]; consequently, what makes the problem difficult is not well understood.

Agarwal et al. [9] first consider the idea of global simplification. They introduce what they call a *weak simplification*: a model in which the vertices of the simplification are not restricted to be a subset of the input vertices, but can lie anywhere in the ambient space.[1] Interestingly, they compare this to a *local* simplification where vertices are restricted to be a subset of the input. We may interpret a combination of two of their results (Theorem 1.2 and Theorem 4.1) as an approximation algorithm for global curve simplification with unrestricted vertices under the Fréchet distance: for a given curve P and threshold δ one can compute, in $O(n \log n)$ time, a simplification P' which has at most the number of vertices of an optimal global simplification with threshold $\delta/8$.

Bereg et al. [24] first explicitly consider global simplification in the setting where vertices are restricted to be a subsequence of input vertices, but using the *discrete Fréchet distance*: a variant of the Fréchet distance which only measures distances between vertices (refer to Section 1.2.2). They show how to compute an optimal

[1] We choose not to adopt the terms *weak* and *strong* in this context because we will also distinguish an intermediate model, and to avoid confusion with the *weak Fréchet* distance; refer to Section 3.2.2.

simplification where vertices are restricted to be a subsequence of the vertices in P in $O(n^2)$ time, and they give an $O(n \log n)$ time algorithm for the setting where vertices may be placed freely in $\mathbb{R}^2$.

Van Kreveld et al. [89] consider the same (global distance, but vertices should be a subsequence) setting, but for the continuous Fréchet and Hausdorff distances. They give polynomial-time algorithms for the Fréchet distance and directed Hausdorff distance (from simplification curve to input curve), but they show the problem is NP-hard for the directed Hausdorff distance in the opposite direction and for the undirected Hausdorff distance. Bringmann and Chaudhury [29] improve their result for the Fréchet distance when the vertices in P' are a subsequence of to $O(n^3)$, and also give a conditional cubic lower bound for curves in high dimensions.

Finally, we mention there is earlier work which does not explicitly study simplification under global distance measures, but contains results that may be interpreted as such. Guibas et al. [67] provide algorithms for computing minimum-link paths that stab a sequence of regions in order. One of the variants, presented in Theorems 10 and 14 of [67], computes what may be seen as an optimal simplification under the Fréchet distance with no vertex restrictions, i.e., the same setting that was studied by Agarwal et al. [9], in $O(n^2 \log^2 n)$ time in in $\mathbb{R}^2$.

In Section 3.2, we present a formal classification of global curve simplification problems. Table 3.1 gives an overview of known results, as well as several new results to complement these (in some cases straightforward adaptations of known results). In the remainder of this chapter we give the main ideas behind our new results shown in this table.

▸ 3.2 Classification of global curve simplification

We aim to provide a systematic overview of curve simplification problems under global distance measures. To this end, we have collected known results and arranged them in a table (Table 3.1), and provided several new results to complement these (refer to Section 3.2.4). This allows us to observe some surprising patterns, and it suggests directions for future research in this important area. We first discuss the dimensions of the table.

▸ 3.2.1 Distance measures

For our study, we consider six different curve distance measures: three variants of the *Hausdorff* distance and three variants of the *Fréchet* distance. These are among the most popular curve distance measures in the algorithms literature. The Haus-

dorff distance captures the distance between the point on one curve that is furthest away to the other curve, and the point on the other curve that is closest to the first point. The variants of the Hausdorff distance we consider are the directed Hausdorff distance from the input to the output, the directed Hausdorff distance from the output to the input, and the undirected (or bidirectional) Hausdorff distance. The Fréchet distance captures the maximum distance between a pair of points traveling along the two curves simultaneously without moving backward. We also consider the weak and discrete Fréchet distance. We have defined these distance measures in Section 1.2.2. We will introduce the rest of the notation for this chapter below.

Let $P = \langle p_1, p_2, \cdots, p_n \rangle$ be the input polygonal curve. Let $\#P$ denote the number of vertices in P. We write $P[s, t]$ for the subcurve between $P(s)$ and $P(t)$ and denote the *shortcut*, i.e., the straight line connecting them, by $\langle P(s)P(t) \rangle$.

▸ 3.2.2 Vertex restrictions

Once we have fixed the distance measure and chosen to apply it globally, one important design decision still remains to be made. Traditional curve simplification algorithms consider the (polygonal) input curve P to be a sequence of points, and produce as output P' a subsequence of this sequence. However, if we measure the distance globally, there may be no strong reason to restrict the family of acceptable output curves so much: the distance measure already ensures the similarity between input and output curves, so we may allow a more free choice of vertex placement. Indeed, several results under this more relaxed viewpoint exist, as discussed in Section 3.1.1. Here, we choose to investigate three increasing levels of freedom: (1) *vertex-restricted* ($\mathcal{V}$), where vertices of P' have to be a subsequence of vertices of P; (2) *curve-restricted (C)*, where vertices of P' can lie anywhere on P but have to respect the order along P; and (3) *non-restricted* ($\mathcal{N}$), where vertices of P' can be anywhere in the ambient space. Figure 3.2 illustrates the difference between the three models. The third category does not make sense for local curve simplification, but is natural for global curve simplification. Observe that when the vertices of a simplified curve have more freedom, the optimal simplified curve never has more, but may have fewer vertices.

▸ 3.2.3 Global curve simplification overview

We are now ready to formally define a class of global curve simplification problems. When $D(\cdot, \cdot)$ denotes a distance measure between curves (e.g., the *Hausdorff* or *Fréchet* distance), the *global curve simplification (GCS)* problem asks for the smallest number k such that there exists a curve P' with at most k vertices, chosen either

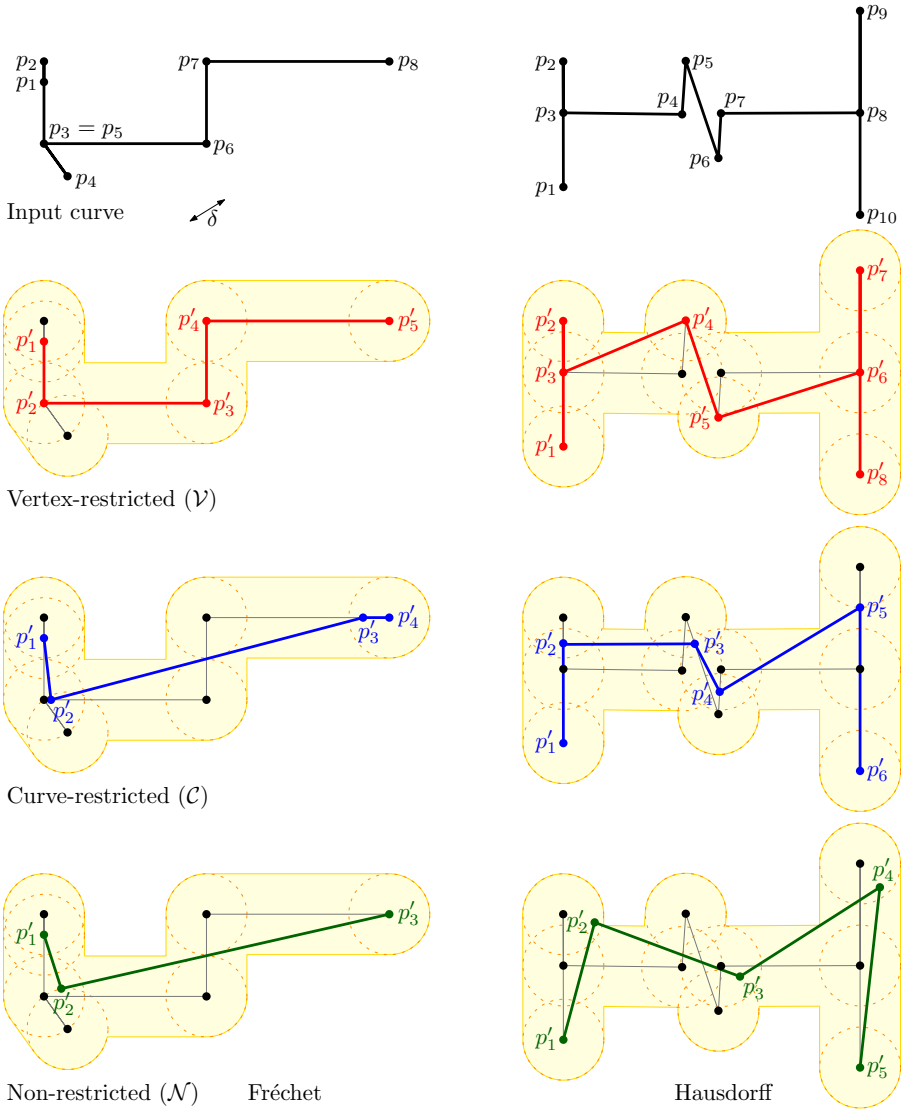


Figure 3.2: The GCS simplified curve in different restrictions under Fréchet (left) and Hausdorff (right) distances. Due to the freedom of choice of placing the vertices of the simplified curve, the number of links monotonically decreases for a fixed input and distance measure.

as a subsequence of the vertices of P (variant $\mathcal{V}$), as a sequence of points on the edges of P in the correct order along P (variant $\mathcal{C}$), or chosen anywhere in $\mathbb{R}^d$ (variant $\mathcal{N}$, unless specified otherwise, d is assumed to be 2), such that $D(P, P') \leq \delta$, for a given threshold δ. In all cases, we require that P and P' start at the same point and end at the same point.

Table 3.1 summarizes results for the different variants of the GCS problem obtained by instantiating D with the Hausdorff or Fréchet distance measures and by applying a vertex restriction R. Here $R \in \{\mathcal{V}, \mathcal{C}, \mathcal{N}\}$, and D is either the undirected Hausdorff distance H, the directed Hausdorff distance $\overleftarrow{\mathsf{H}}$ from P to P', the directed Hausdorff distance $\overrightarrow{\mathsf{H}}$ from P' to P, the Fréchet distance F, the discrete Fréchet distance dF, or the weak Fréchet distance wF. Throughout the chapter we use $D_R(P, \delta)$ to denote a curve P' that is the optimal R-restricted global simplification of P with $D(P, P') \leq \delta$.

Although this chapter focuses on min-link simplification, for approximation algorithms it can be interesting to have a simplification that approximates optimal values for both the number of links and the distance δ at the same time. We call an algorithm an (α, β)-*approximation* if it computes a solution with distance at most $\beta\delta$ and uses at most α times more links than the optimal solution for δ.

▸ 3.2.4 New results

In order to provide a thorough understanding of the different variants of the GCS problem we provide several new results. In some cases these are straightforward adaptations of known results, in other cases they require deeper ideas. In Section 3.3.3, we give polynomial time algorithms for the vertex-restricted GCS problem under the strong Fréchet (Theorem 3.3.8) and weak Fréchet (Theorem 3.3.2) distances.[2] In Section 3.4 we show that the vertex-restricted problem under the directed Hausdorff distance from P' to P considered by Van Kreveld et al. [89] can be improved to $O(n^3 \log n)$ time (Theorem 3.4.2).

In Section 3.5 we prove that solving the curve-restricted GCS problem is NP-hard for almost all measures considered in this chapter except for the discrete Fréchet distance (Theorem 3.6.3) and Fréchet distance in $\mathbb{R}^1$ (Theorem 3.7.1) for which we present polynomial time algorithms. Surprisingly the Fréchet distance computation is slower than the GCS problem under the Fréchet distance in 1D un-

[2] An algorithm with a running time of $O(n^4)$ for the vertex-restricted variant under the strong Fréchet distance is presented in Theorem 3.3.5. Bringmann and Chaudhury [29] independently developed an $O(n^3)$ algorithm for the same problem. Our current algorithm in Section 3.3.3 uses a vital insight from [29] to improve our $O(n^4)$ algorithm to $O(n^3)$ as well. The resulting algorithm uses less space than the algorithm in [29].

Table 3.1: Known and new results for the GCS problem under global distance measures.

Distance	Vertex-restricted ($\mathcal{V}$)	Curve-restricted ($\mathcal{C}$)	Non-restricted ($\mathcal{N}$)
$\overleftarrow{\mathsf{H}}$	strongly NP-hard [89]	weakly NP-hard (Thm 3.5.10)	strongly NP-hard (Thm 3.10.8)
$\overrightarrow{\mathsf{H}}$	$O(n^4)$ [89] $O(n^3 \log n)$ (Thm 3.4.2)	weakly NP-hard (Thm 3.5.8)	poly(n) [88]
H	strongly NP-hard [89]	strongly NP-hard (Cor 3.9.2)	strongly NP-hard (Thm 3.9.1)
F	$O(mn^5)$ [89] $O(n^3)$ (Thm 3.3.8) $O(n^3)$ [29]	$O(n)$ in $\mathbb{R}^1$ (Thm 3.7.1) weakly NP-hard in $\mathbb{R}^2$ (Thm 3.5.6)	$O(n^2 \log^2 n)$ in $\mathbb{R}^2$ [67] $O(n \log n)$ $(1, 8)$-approx [9] $O^*(n^2 \log n \log\log n)$ $(2, 1+\varepsilon)$-approx (Thm 3.8.8)
dF	$O(n^2)$ [24]	$O(n^3)$ (Thm 3.6.3)	$O(n \log n)$ [24]
wF	$O(n^3)$ (Thm 3.3.2)	weakly NP-hard (Thm 3.5.10)	$O^*(n^2 \log n \log\log n)$ $(2, 1+\varepsilon)$-approx (Cor 3.8.9)

like the Hausdorff distance under which the simplification becomes harder than the distance computation. To the best of our knowledge, these are the first results in the curve-restricted setting under global distance measures. In Section 3.8, we give $(2, 1+\varepsilon)$-approximation algorithm for computing $\mathsf{F}_{\mathcal{N}}(P, \delta)$ which runs in $O^*(n^2 \log n \log\log n)$ time where O^* hides factors polynomial in $1/\varepsilon$, for any $0 < \varepsilon \leq 1$ (Theorem 3.8.8). We also argue that the same result holds for computing $\mathsf{wF}_{\mathcal{N}}(P, \delta)$ (Corollary 3.8.9). In Section 3.9 we show that this problem becomes NP-hard when we consider the Hausdorff distance, i.e., $\mathsf{H}_{\mathcal{N}}(P, \delta)$ (Theorem 3.9.1). Finally, in Section 3.10 we show NP-hardness for computing $\overleftarrow{\mathsf{H}}_{\mathcal{N}}(P, \delta)$.

▸ 3.2.5 Discussion

With both the existing work and our new results in place, we now have a good overview of the complexity of the different variants of the GCS problem, see Table 3.1. Observe that the curve-restricted variants appear to often be harder than both the vertex-restricted and the non-restricted variants. That means that, on the one hand, broadening the search space from the vertex-restricted to the curve-restricted case makes the problem harder. But on the other hand it does not give unrestricted freedom of choice, which in turn enables the development of efficient algorithms for the unrestricted case.

Another interesting pattern can be observed for the Hausdorff distance measures. The direction of the Hausdorff distance makes a significant difference in whether the corresponding GCS problem is NP-hard or polynomially solvable. The GCS problem for the undirected Hausdorff distance is at least as hard as for the directed Hausdorff distance from the input curve to the simplification. Drawing upon the above observations we make the following conjecture:

Conjecture 1. *The curve-restricted GCS problem for $\overleftarrow{\mathsf{H}}$ is strongly NP-hard.*

Also, note that we only prove several problem variants to be NP-hard. The question of whether these problems are also in NP remains open. It is interesting to note that for the problem of computing a minimum-link path inside a simple polygon, it has been shown that coordinates with exponential bit complexity are sometimes required [88]. This suggests the problems we discuss may have a similar structure.

▸ 3.3 Vertex-restricted simplification under Fréchet distance

We use the free space diagram between P and its shortcut graph G to solve the vertex-restricted GCS problem under the weak and strong Fréchet distances in $O(n^3)$ time and space. This is related to *map-matching* [15], however in our case we need to compute *shortest* paths in the free space that correspond to *simple* paths in G. While map-matching for closed simple paths is NP-complete [107], we exploit the DAG property of G to develop efficient algorithms.

▸ 3.3.1 Shortcut DAG and free space diagram

For a given polygonal curve P, we define its *shortcut DAG* $G = G(P) = (V, E)$, where $V = \{1, \dots, n\}$ and $E = \{(u, v) \mid 1 \le u < v \le n\}$. We consider each $v \in V$ to be embedded at p_v and each edge $e = (u, v) \in E$ to be embedded as a straight line shortcut linearly parameterized as $e(t) = (1 - t)p_u + tp_v$ for $t \in [0, 1]$. We consider the parameter space of G to be $E \times [0, 1]$.

We described the *free space diagram* for computing the Fréchet distance between polygonal curves in Section 1.2.2. In this chapter, we also make use of a generalized version that is defined between a polygonal curve and a graph, specifically, the shortcut DAG.

Let $\delta > 0$, and consider the joint parameter space $[1, n] \times E \times [0, 1]$ of P and G. Any $(s, e, t) \in [1, n] \times E \times [0, 1]$ is *free* in the diagram if $\|P(s) - e(t)\| \le \delta$. For brevity, we write $(s, e(t))$ instead of (s, e, t), and if $e(t) = v \in V$ we write (s, v). The *free space diagram* $\mathsf{FSD}_\delta(P, G)$ consists of all points in $[1, n] \times E \times [0, 1]$ together with an annotation for each point whether it is free or not. See Figure 1.2 for an example of a free space diagram between two curves. The free space diagram $\mathsf{FSD}_\delta(P, G)$ consists of one *cell* for each edge in P and each edge in G. The free space in such a cell is convex. The boundary of a cell consists of four line segments which each contain at most a single *free space interval*. We consider the free space diagram to be composed of spines and strips; see Figure 3.3. For any $v \in V$ and $e \in E$ we call $\mathsf{SP}(v) = [1, n] \times v$ a *spine* and $\mathsf{ST}(e) = [1, n] \times e \times [0, 1]$ a *strip*. We denote the free space within spines and strips as $\mathsf{SP}_\delta(v) = \{(s, v) \mid 1 \le s \le n,\ \|P(s) - p_v\| \le \delta\}$ and $\mathsf{ST}_\delta(e) = \{(s, e(t)) \mid 1 \le s \le n,\ 0 \le t \le 1,\ \|P(s) - e(t)\| \le \delta\}$, respectively. For any edge $(u, v) \in E$, both spines centered at the vertices of the edge are subsets of the strip: $\mathsf{SP}(u), \mathsf{SP}(v) \subseteq \mathsf{ST}(u, v)$, and $\mathsf{SP}(u)$ is a subset of all strips with respect to edges incident on u.

▸ 3.3.2 Free space based algorithm for weak Fréchet simplification

In this section we provide an algorithm to solve the vertex-restricted GCS problem under the weak Fréchet distance in $O(n^3)$ time and space. We will see that the optimal simplification P' is a shortest simple path in G that corresponds to a path $\mathcal{P}$ from $(1,1)$ to (n,n) in $\mathsf{FSD}_\delta(P,G)$ that is contained in free space.

Let $P' = \mathsf{wF}_\mathcal{V}(P,\delta)$ be an optimal vertex-restricted simplification of P and let $n' = \#P'$. Then P' is a path in G, and P' visits an increasing subsequence of vertices in P (or V). From the fact that $\mathsf{wF}(P,P') \leq \delta$ we know that there is a path $\mathcal{P} = (\sigma,\theta)$ from $(1,1)$ to (n,n') in $\mathsf{FSD}_\delta(P,P')$ that lies entirely within free space. Since $\mathsf{FSD}_\delta(P,P')$ is a subsequence of $\mathsf{FSD}_\delta(P,G)$, the path $\mathcal{P} = (\sigma,\theta)$ is also a path in $\mathsf{FSD}_\delta(P,G)$. Here, σ is a reparameterization of P, and θ is a reparameterization of a *simple* path $P' = \langle p_{i_1}, p_{i_2}, \dots, p_{i_k}\rangle$ in G with $i_1 = 1$ and $i_k = n$. We call (s,d) in $\mathsf{FSD}_\delta(P,G)$ *weakly reachable* if there exists a path $\mathcal{P} = (\sigma,\theta)$ from $(1,1)$ to (s,d) in $\mathsf{FSD}_\delta(P,G)$ that lies in free space such that θ is a reparameterization of a simple path from p_1 to some point on an edge in G. We denote the number of spines hit by this simple path $\mathcal{P}$ by $\#\mathcal{P}$, and we call $\mathcal{P}$ *weakly reachable*. We define the cost function $\phi : [1,n]\times V \longrightarrow \mathbb{N}\cup\{0\}$ as $\phi(z,v) = \min_\mathcal{P} \#\mathcal{P}$, where the minimum ranges over all weakly reachable paths to (z,v) in the free space diagram. If no such path exists then $\phi(z,v) = \infty$. Note that all points in a free space interval (on the boundary of a free space cell) have the same ϕ-value.

Observation 3.3.1. *There is a weakly reachable path $\mathcal{P}$ in $\mathsf{FSD}_\delta(P,G)$ from $(1,1)$ to (n,n) with $\#\mathcal{P} = \#\mathsf{wF}_\mathcal{V}(P,\delta)$ if and only if $\phi(n,n) = \#\mathsf{wF}_\mathcal{V}(P,\delta)$.*

Since $\phi(n,n) = \#\mathsf{wF}_\mathcal{V}(P,\delta)$ is the length of a shortest simple path P' in G' that

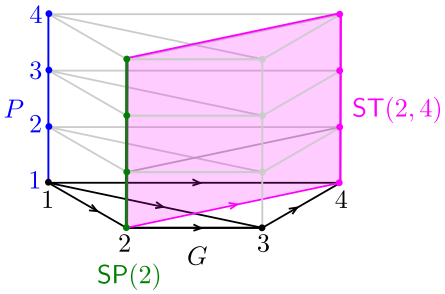


Figure 3.3: Spines and strips in a free space surface.

corresponds to a weakly reachable path $\mathcal{P} = (\sigma, \theta)$ in $\mathsf{FSD}_\delta(P, G)$, we can compute $\phi(n, n)$ by propagating ϕ-values across free space intervals in a breadth-first manner. However, we need to construct both P' and $\mathcal{P}$ during our algorithm. As opposed to generic map-matching algorithms for the weak Fréchet distance [28], we need to ensure that P' is simple.

Since ϕ is the length of a shortest path, it seems as if one could compute it by simply using a breadth-first propagation. However, one has to be careful because a weakly reachable path $\mathcal{P}$ is only allowed to backtrack *along the path in G that it has already traversed*. We therefore carefully combine two breadth-first propagations to compute the ϕ values for all $I \in \mathcal{I}$, where $\mathcal{I}$ is the set of all (non-empty) free space intervals on all spines $\mathsf{SP}(v)$ for all $v \in V$. For the primary breadth-first propagation, we initialize a queue Q by enqueuing the interval $I \subseteq \mathsf{SP}_\delta(1)$ that contains $(1, 1)$. Once an interval has been enqueued it is considered *visited*, and it can never become unvisited again. Then we repeatedly extract the next interval I from Q. Assume $I \subseteq \mathsf{SP}_\delta(u)$. For each v from $u+1$ to n we consider $\mathsf{ST}(u, v)$ and we compute all unvisited intervals $J \subseteq \mathsf{SP}_\delta(u) \cup \mathsf{SP}_\delta(v)$ that are reachable from I with a path in $\mathsf{ST}_\delta(u, v)$. These J can be reached using one more vertex, therefore we set $\phi(J) = \phi(I) + 1$, we insert J into Q, and we store the predecessor $\pi(J) = I$. For each $J \in \mathsf{SP}_\delta(u)$ we then launch a secondary breadth-first traversal to propagate $\phi(J)$ to all unvisited intervals $J' \in \mathcal{I}$ that are reachable from J within the free space of $\mathsf{FSD}_\delta(P, G(\pi(J)))$. Here, $G(\pi(J))$ denotes the projection of the predecessor DAG rooted at $\pi(J)$ onto G, i.e., each interval I in the predecessor DAG is projected to u if $I \subseteq \mathsf{SP}_\delta(u)$. This allows $\mathcal{P}$ to backtrack along the path in G that it has already traversed, *without* increasing ϕ. This secondary breadth-first traversal uses a separate queue Q', and sets $\phi(J') = \phi(J)$ and $\pi(J') = J$. When this secondary traversal is finished, Q' is prepended to Q, and then the primary breadth-first propagation continues. Once Q is empty, i.e., all intervals have been visited, $\phi(I) = \#\mathsf{wF}_\mathcal{V}(P, \delta)$, where $I \subseteq \mathsf{SP}_\delta(n)$ is the interval that contains (n, n). Backtracking a path from n to 1 in the predecessor DAG $\pi(I)$ yields the simplified curve P'. This algorithm visits each interval in $\mathcal{I}$ once using nested breadth-first traversals. Since there are $O(n^3)$ free space intervals this takes $O(n^3)$ time and space.

Theorem 3.3.2. *Let P be polygonal curve in $\mathbb{R}^d$ with n vertices and $\delta > 0$ be a real. One can compute $\mathsf{wF}_\mathcal{V}(P, \delta)$ in $O(n^3)$ time and space.*

▸ 3.3.3 Extended algorithm for Fréchet distance simplification

In this section we provide an algorithm to solve the vertex-restricted GCS problem under the Fréchet distance.

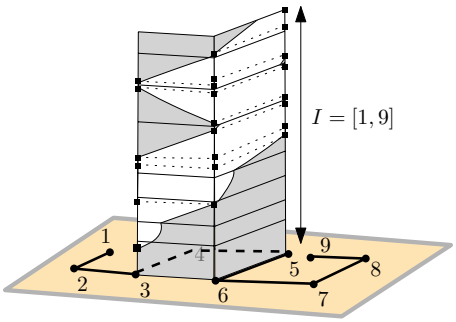


Figure 3.4: Elementary intervals on each spine are created by subdividing free space intervals with all endpoints of free space intervals on all other spines.

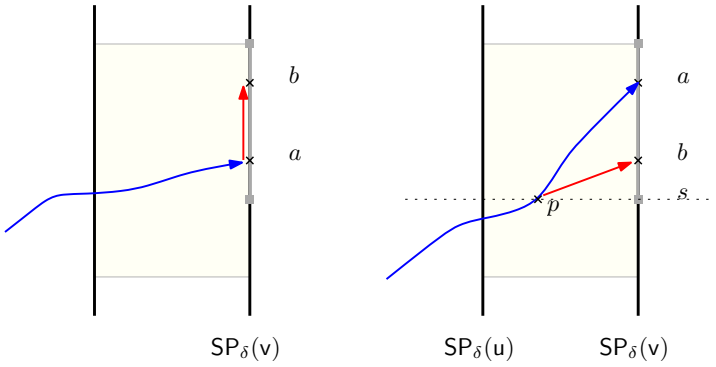
Elementary intervals and cost function Φ Let $S \subseteq [1, n]$ be the set of all interval endpoints of all free space intervals of $\mathsf{SP}_\delta(v) \subseteq \mathsf{SP}(v) = [1, n] \times v$ for all $v \in V$, projected onto $[1, n]$. The set S induces a partition of $[1, n]$ into intervals; these are half-open $[a, b)$, except for the last interval $[a, b]$ which is closed. For each $v \in V$, let $\mathsf{L}_\delta(v)$ be the ordered list of *elementary intervals* obtained by subdividing the intervals of $\mathsf{SP}_\delta(v)$ according to this partition; see Figure 3.4. We assume that elementary intervals in $\mathsf{L}_\delta(v)$ are ordered in increasing order of their starting point. Projecting all $\mathsf{L}_\delta(v)$ to $[1, n]$ induces a total order on all elementary intervals; we use $<$ and $=$ to compare intervals.

We define the cost function $\phi : [1, n] \times V \longrightarrow \mathbb{N} \cup \{0\}$ as $\phi(z, v) = \min_{\mathcal{P}} \#\mathcal{P}$, where the minimum ranges over all reachable paths from $(1, 1)$ to (z, v) in the free space diagram. Here, $\mathcal{P}$ is called *reachable* if it starts in $(1, 1)$, lies entirely in free space, and is monotone in P and on each edge in E. If no such path exists then $\phi(z, v) = \infty$. The function ϕ is defined on all spines $\mathsf{SP}_\delta(v)$ for all $v \in V$, and it captures min-link reachability in the free space diagram. We will propagate ϕ across the free space diagram using dynamic programming.

Lemma 3.3.3 (Properties of elementary intervals)**.** *Let $v \in V$. Then:*

1. $|\mathsf{L}_\delta(v)| \leq 2n^2 + 1$.
2. $\phi(a, v) = \phi(b, v)$ *for all* $a, b \in e \in \mathsf{L}_\delta(v)$.

Proof. 1. There are at most n free space intervals on each spine, and there are n spines, therefore there are $|S| = 2n^2$ interval endpoints. These subdivide $[1, n]$ into $2n^2 + 1$ intervals.



(a) (b)

Figure 3.5: Illustration of the proof of property 2. a and b belong to the same elementary interval e that is highlighted in gray. (a) $b > a$ (b) if $a > b$.

2. For the sake of contradiction, assume there exist $a, b \in e$, with $a \neq b$ such that $\phi(a, v) \neq \phi(b, v)$. Assume without loss of generality that $\phi(a, v) < \phi(b, v)$. Let $\mathcal{P}(a, v)$ be a reachable path to (a, v) with $\#\mathcal{P}(a, v) = \phi(a, v)$.

2a. If $b > a$, then extending $\mathcal{P}(a, v)$ to continue vertically from a to b along e yields a reachable path $\mathcal{P}(b, v)$ to (b, v) with $\#\mathcal{P}(b, v) = \#\mathcal{P}(a, v)$; see Figure 3.5(a). Therefore $\phi(b, v) \leq \phi(a, v)$ which is a contradiction.

2b. Now, assume that $a > b$. Let s be the start point of e, and let p be the last point on $\mathcal{P}(a, v)$ such that $p = (s, j)$ for some $u \leq j < v$ with $u \in V$. Then p lies in some strip $\mathsf{ST}_\delta(u, v)$. Since $p, (a, v), (b, v)$ all lie in free space and e is a subset of a free space interval of a single cell, $p, (a, v), (b, v)$ all lie in a single free space cell. Hence we p with a line segment to (b, v), which yields a reachable path $\mathcal{P}(b, v)$ with $\#\mathcal{P}(b, v) = \#\mathcal{P}(a, v)$; see Figure 3.5(b). Therefore $\phi(b, v) \leq \phi(a, v)$ which is a contradiction.

□

By Lemma 3.3.3, ϕ is constant on each elementary interval $e \in \mathsf{L}_\delta(v)$. For brevity we write $\phi(e, v)$ and we write $\mathcal{P}(e, v)$ for a reachable path to e. Lemma 3.3.4 below shows the recursive formula for ϕ which we will use in our dynamic programming algorithm.

Lemma 3.3.4 (Recursive formula)**.**

1. *For all $e \in \mathsf{L}_\delta(1)$: If e is reachable then $\phi(e,1) = 0$, otherwise $\phi(e,1) = \infty$.*
2. *For all $v \in \{2,\dots,n\}$ and $e \in \mathsf{L}_\delta(v)$: $\phi(e,v) = \min\limits_{1\le u<v} \min\limits_{\substack{e' \in \mathsf{L}_\delta(u) \\ e' \le e}} \phi(e',u) + 1$,*

   where the second minimum is taken over all e' such that there exists a reachable path from e' to e within $\mathsf{ST}_\delta(u,v)$.

Proof. 1. If e is reachable, then we know that $\phi(e,1) = \#\mathcal{P}(e,1) = 0$ because e belongs to the first spine $\mathsf{L}_\delta(1)$. If e there is no reachable path to e then $\phi(e,1) = \infty$.

2. By definition and Lemma 3.3.3, $\phi(e,v) = \min_{\mathcal{P}(e,v)} \#\mathcal{P}(e,v)$, where the minimum ranges over all reachable paths to (e,v) in the free space diagram, and reachable paths are monotone in P and on each edge in E.

Let $\mathcal{P}^*(e,v)$ be a reachable path to (e,v) such that $\#\mathcal{P}^*(e,v) = \phi(e,v)$. Now consider the last spine $\mathsf{SP}(u)$ that $\mathcal{P}^*(e,v)$ visits before visiting $\mathsf{SP}(v)$, and let $e' \in \mathsf{L}_\delta(u)$ be the last elementary interval it visits. Let $\mathcal{P}_1$ be the sub-path of $\mathcal{P}^*(e,v)$ that ends in e' on $\mathsf{SP}(u)$, and let $\mathcal{P}_2$ be the remaining portion of $\mathcal{P}^*(e,v)$ that starts in e' on $\mathsf{SP}(u)$ and ends in e on $\mathsf{SP}(v)$. Then $\mathcal{P}_2$ is a reachable path from (e',u) to (e,v), and due to monotonicity in E, $\mathcal{P}_2$ has to lie in $\mathsf{ST}(u,v)$. Therefore $\#\mathcal{P}_2 = 1$ and $\#\mathcal{P}^*(v,e) = \#\mathcal{P}_1 + \#\mathcal{P}_2 = \#\mathcal{P}_1 + 1$ by construction. We know that $\#\mathcal{P}_1$ has to be the minimum number of links to reach (e',u), because otherwise $\#\mathcal{P}(e,v)$ would not be optimal. Hence, $\phi(e',u) = \#\mathcal{P}_1$ and we have $\phi(e,v) = \phi(e',u) + 1$.

Since edges in E are directed, any reachable path to (e,v) can only visit spines $\mathsf{SP}(u)$ for $u \in \{1,\dots,v\}$. Any such path has to be monotone in P and therefore can only visit elementary intervals $e' < e$. Thus, $\phi(e,v) = \min\limits_{1\le u<v} \min\limits_{\substack{e' \in \mathsf{L}_\delta(u) \\ e' \le e}} \phi(e',u) + 1$, where only those e' are considered for which there is a reachable path from e' to e in $\mathsf{ST}(u,v)$. □

DP algorithm First we compute the shortcut DAG G and the free space diagram $\mathsf{FSD}_\delta(P,G)$. This can be done by computing free space diagrams for all strips (and spines) and connecting them together with respect to the adjacency information in G. For every $v \in V$ we then compute the list $\mathsf{L}_\delta(v)$ of all elementary intervals in $\mathsf{SP}(v)$, and we initialize $\phi(v,e)$ according to Lemma 3.3.4. Algorithm 1 processes the free space diagram spine by spine for $v \in \{2,\dots,n\}$.

For fixed v we compute $\phi(e,v)$ for all $v \in \mathsf{L}_\delta(v)$ by propagating reachability according to the recursive formula in Lemma 3.3.4 using the following batching approach; refer to Figure 3.6 for an illustration. For all $u < v$ and all elementary intervals $e' \in \mathsf{L}_\delta(u)$ we compute the *reachable interval* $I_v(e') \subseteq \mathsf{SP}(v)$ which is the

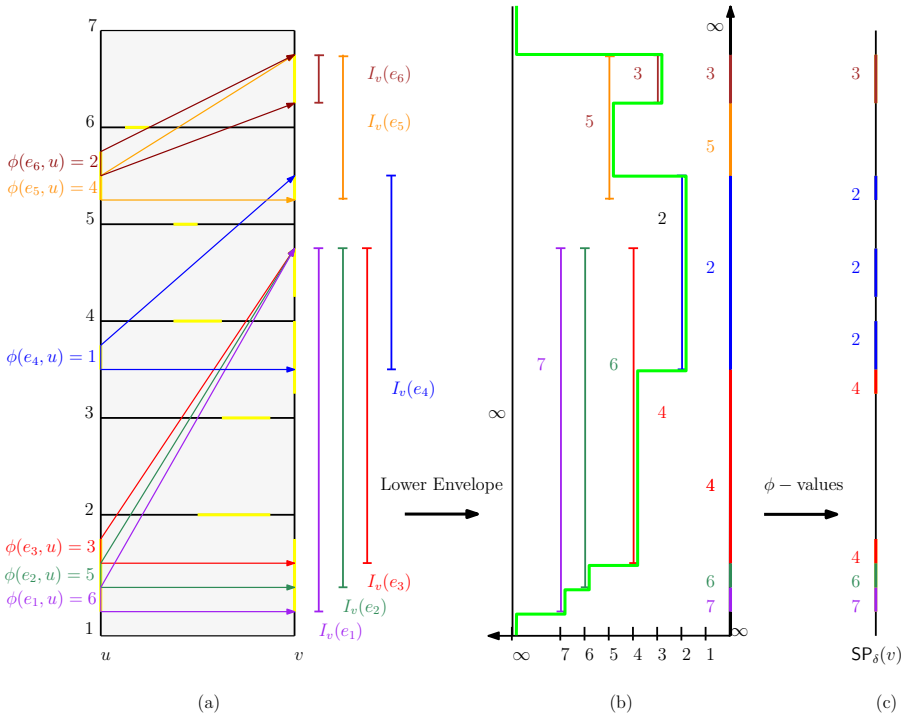


Figure 3.6: Propagating ϕ within $\mathsf{ST}_\delta(u, v)$. (a) shows reachable intervals on $\mathsf{SP}(v)$ computed for elementary intervals in $\mathsf{L}_\delta(u)$. (b) shows the computation of the lower envelope $\mathcal{L}_u$ of all reachable intervals $I_v(e')$ for all $e' \in \mathsf{L}_\delta(u)$. (c) shows the resulting ϕ-values.

smallest (continuous) interval in $\mathsf{SP}(v)$ that includes all points $z \in \mathsf{SP}_\delta(v)$ that are reachable from e' within $\mathsf{ST}_\delta(u, v)$. Note that $I_v(e')$ generally includes points from multiple elementary intervals as well as non-reachable points. For all points $z \in I_v(e')$ that can be reached with a path that visits e' and traverses $\mathsf{ST}_\delta(u, v)$, we know that $\phi(z, v) \geq \phi(e', u) + 1$. We therefore associate the cost $\psi(I_v(e')) = \phi(e', u) + 1$ with the reachable interval $I_v(e')$. Thus, $\psi(I_v(e'))$ is a partially defined constant function on $\mathsf{SP}(v)$, the graph of which is a line segment $I_v(e') \times \psi(I_v(e'))$. For each $u < v$, we compute the *lower envelope* $\mathcal{L}_u : \mathsf{SP}(v) \longrightarrow \mathbb{R}$ of all line segments $I_v(e') \times \psi(I_v(e'))$ for all $e' \in \mathsf{L}_\delta(u)$, which is the pointwise minimum of all these partially defined constant functions. For each elementary interval $e \in \mathsf{L}_\delta(v)$, we have that $\mathcal{L}_u$ is constant by construction, and we compute $\phi(e, v) = \min_{1 \leq u < v} \mathcal{L}_u(e)$, which follows from the recursive formula in Lemma 3.3.4. After having processed all $v \in \{2, \dots, n\}$, all $\phi(e, v)$ have been computed. We then construct a reachable path $\mathcal{P}((1, 1), (n, n))$ by backtracking the ϕ-values from $\phi(n, n)$. The sequence of traversed strips corresponds to the desired path P' in G.

There are a total of $O(n^3)$ elementary intervals. For fixed $u < v$, using Lemma 3 of [15] we can compute all reachable intervals within a single strip $\mathsf{ST}(u, v)$ in $O(|\mathsf{L}_\delta(u)|) = O(n^2)$ time. Each lower envelope $\mathcal{L}_u$ can be computed in time linear in the number of intervals, hence $O(n^2)$ time. Computing $\phi(e, v) = \min_{1 \leq u < v} \mathcal{L}_u(e)$ for all $e \in \mathsf{L}_\delta(v)$ takes $O(n^3)$ time, thus the total runtime is $O(n^4)$ using $O(n^3)$ space.

Theorem 3.3.5. *Given a polygonal curve P with n vertices and $\delta > 0$, an optimal solution to the vertex-restricted GCS problem under Fréchet distance can be computed in $O(n^4)$ time and $O(n^3)$ space.*

Proof. The main task is to compute $\phi(n, n)$. The shortcut DAG can be computed straight-forwardly in $O(n^2)$ time and space, and the free space diagram requires computing all spines and strips. There are $O(n^2)$ strips and it takes $O(n)$ time and space to compute the free space diagram for each strip, for a total of $O(n^3)$ time and $O(n^2)$ space. By Lemma 3.3.3 there are $O(n^2)$ elementary intervals per spine and thus $O(n^3)$ elementary intervals total, and thus all the $\mathsf{L}_\delta(v)$ can be computed in $O(n^3)$ time. The ϕ-value of each of the $O(n^3)$ elementary intervals is initialized in lines 5 and 6 in overall $O(n^3)$ time.

Lines 7-14 compute all $\phi(e, v)$ for all $e \in \mathsf{L}_\delta(v)$ by batching the reachability propagations according to the recursive formula in Lemma 3.3.4. Lemma 3 of [15] shows that within a single strip $\mathsf{ST}(u, v)$, all reachable intervals for all $O(n)$ free space intervals can be computed in $O(n)$ time. In our case, for fixed $u < v$, we have $O(n^2)$ elementary intervals in $\mathsf{L}_\delta(u)$ (which are subdivisions of the free space intervals), and we can apply this lemma to compute all their reachable intervals in $O(n^2)$ time.

Algorithm 1: Algorithm for computing $\mathsf{F}_{\mathcal{V}}(P, \delta)$.

Input : Polygonal curve $P = \langle p_1, p_2, \cdots, p_n \rangle$, and $\delta > 0$
Output : Optimal vertex-restricted simplification P' of P such that $\mathsf{F}(P, P') \leq \delta$

1 Compute shortcut DAG $G = (V = \{1, \ldots, n\}, E)$
2 Compute $\mathsf{FSD}_\delta(P, G)$
3 **for all** $v \in V$ **:**
4 Compute the list of elementary intervals $\mathsf{L}_\delta(V)$
5 **for all** $e \in \mathsf{L}_\delta(v)$ **:** $\phi(v, e) = \infty$ *// Initialize ϕ*
6 **for all** $e \in \mathsf{L}_\delta(1)$ *that is reachable from* $(1, 1)$ **:** $\phi(1, e) = 0$ *// Initialize ϕ*
7 **for** $v = 2$ **to** n **:**
8 **for all** $u \in \{1, \cdots, v - 1\}$ **:**
9 $S = \emptyset$
10 **for all** $e' \in \mathsf{L}_\delta(u)$ **:**
11 Compute the reachable interval $I_v(e')$ and set $\psi(I_v(e')) = \phi(e', u) + 1$
12 $S = S \cup \{I_v(e')\}$
13 Compute the lower envelope $\mathcal{L}_u$ of all $I \times \psi(I)$ for all $I \in S$
14 **for all** $e \in \mathsf{L}_\delta(v)$ **:** $\phi(e, v) = \min_{1 \leq u < v} \mathcal{L}_u(e)$
15 Return vertices of P' by tracing back the ϕ values.

We assign to each reachable interval $I_v(e)$ the cost ψ that it takes to reach e with a min-link path through e'. For fixed v and u, lines 10-12 take $O(n^2)$ time. Processing elementary intervals $e \in \mathsf{L}_\delta(u)$ in increasing order, yields reachable intervals in increasing order of their start points. This allows us to compute the lower envelope $\mathcal{L}_u : \mathsf{SP}_\delta(v) \longrightarrow \mathbb{R}$ in linear time in the number of intervals (see Section 5 of [20]), hence in $O(n^2)$ time. By construction, $\mathcal{L}_u$ is constant on each $e \in \mathsf{L}_\delta(v)$. From the recursive formula in Lemma 3.3.4, follows that $\phi(e, v) = \min_{1 \leq u < v} \mathcal{L}_u(e)$, which is computed in $O(n^3)$ time in line 14. Therefore the entire runtime is $O(n^2 \cdot n^2 + n \cdot n^3) = O(n^4)$, and the overall space needed is $O(n^3)$. A min-link reachable path in the free space diagram can be traced back from $\phi(n, n)$ in $O(n^3)$ time, and the sequences of spines visited corresponds to the simplification P'. □

Improved DP algorithm We now describe an improvement of the algorithm to $O(n^3)$ time. The runtime bottleneck of the algorithm is that, for fixed $v \in V$, the propagation step which computes all $\phi(e, v)$ for all $e \in \mathsf{L}_\delta(v)$, takes $O(n^3)$ time, even

though there are only $|\mathsf{L}_\delta(v)| = O(n^2)$ elementary intervals. It turns out that the ψ-values of the reachable intervals have a special structure that we can exploit to speed up this propagation to take only $O(n^2)$ time.

For this we need to consider portions of strips and spines: Let $u < v$. We define $\mathsf{ST}^{[i,j]}(u,v) = [u,v]\times[i,j]$ and $\mathsf{SP}^{[i,j]}(u) = u\times[i,j]$. Let $\mathsf{L}_\delta^{[i,j]}(u) = \mathsf{L}_\delta(u)\cap\mathsf{SP}^{[i,j]}(u)$. For each elementary interval $e \in \mathsf{L}_\delta^{[i,j]}(u)$, we break the reachable interval $I_v(e) \subseteq \mathsf{SP}(v)$ into the *upper reachable interval* $I_v^\top(e) = I_v(e) \cap \mathsf{SP}^{[i+1,n]}(v)$ and the *lower reachable interval* $I_v^\perp(e) = I_v(e) \cap \mathsf{SP}^{[i,i+1]}(v)$.

Observation 3.3.6. *Let $e, e' \in \mathsf{L}_\delta^{[i,i+1]}(u)$. Then:*

- *$I_v^\top(e)$ and $I_v^\top(e')$ are identical.*
- *$I_v^\perp(e) = [a, i]$, where a is the maximum of the bottom endpoints of e and the free space interval on $\mathsf{SP}_\delta^{[i,i+1]}(u)$, or ∞ if the free space interval is empty.*

This means that all elementary intervals in $\mathsf{L}_\delta^{[i,i+1]}(u)$ generate the same upper reachable interval, and lower reachable intervals depend on bottom endpoints only.

Lemma 3.3.7 (Monotonicity of Φ)**.** *Let $e_1 < \cdots < e_m$ be all elementary intervals in $\mathsf{L}_\delta^{[i,i+1]}(u)$. Then $\phi(e_m, u) \le \ldots \le \phi(e_2, u) \le \phi(e_1, u)$.*

Proof. We use proof by contradiction. Let j and k be two integers such that $1 \le j < k \le m$, $e_j < e_k$ and $\phi(e_k, v) > \phi(e_j, v)$. Then a reachable path $\mathcal{P}(e_j, 1)$ of length $\#\mathcal{P}(e_j, 1) = \phi(j, v)$ is also reaching e_k since all $e_1, \cdots, e_m$ are within free space interval of $[i, i+1] \cap \mathsf{SP}_\delta(v)$. Therefore, $\#\mathcal{P}(e_k, 1) = \#\mathcal{P}(e_j, 1) = \phi(e_j, v)$ and we have $\phi(e_k, v) \le \phi(e_j, v)$ which is a contradiction. □

Now let $\mu^{[i,i+1]}(v)$ be the minimum ϕ-value in $\mathsf{L}_\delta^{[i,i+1]}(v)$. For each $e \in \mathsf{L}_\delta^{[i,i+1]}(v)$ let

$$\bar\phi(e, v) = \min_{1\le u<v} \min_{\substack{e'\in\mathsf{L}_\delta^{[i,i+1]}(u)\\ e'\le e}} \phi(e', u) + 1 = \min_{\substack{e'\in\mathsf{L}_\delta^{[i,i+1]}(v-1)\\ e'\le e}} \bar\phi(e', v-1) + 1\ ,$$

where only those e' are considered for which there exists a reachable path from e' to e within $\mathsf{ST}_\delta^{[i,i+1]}(u, v)$. The function $\bar\phi$ propagates horizontal reachability within $[i, i+1]$ only.

The propagation now proceeds in two steps, for fixed v:

1. For all $u < v$ and all $1 \le i < n$ we compute the upper interval $I_v^\top(e')$ for any fixed $e' \in \mathsf{L}_\delta^{[i,j]}(u)$ using Lemma 3 of [15], and we know that $\psi(I_v^\top(e')) =$

$\mu^{[i,i+1]}(u) + 1$. This generates $O(n^2)$ upper reachable intervals. We compute their lower envelope $\mathcal{L}^1$.

2. We propagate lower reachable intervals *only from* $u = v - 1$ *using* $\bar{\phi}$: For all $e' \in \mathsf{L}_\delta(v-1)$ we compute the lower interval $I_v^\perp(e')$. This can be done in constant time per interval, and it takes $O(|\mathsf{L}_\delta(v-1)|) = O(n^2)$ time. We set $\psi(I_v^\top(e')) = \bar{\phi}(e') + 1$. We compute their lower envelope $\mathcal{L}^2$. Finally we need to update ϕ using the lower envelope of $\mathcal{L}^1$ and $\mathcal{L}^2$ and we also update $\bar{\phi}$ using $\mathcal{L}^2$ only.

Thus the runtime for an update step is $O(n^2)$ and the total runtime is $O(n^3)$.

Theorem 3.3.8. *Let P be a polygonal curve with n vertices and let $\delta > 0$ be a real value. One can compute $\mathsf{F}_\mathcal{V}(P, \delta)$ in $O(n^3)$ time and $O(n^3)$ space.*

▸ 3.4 Computing $\overrightarrow{\mathsf{H}}_\mathcal{V}(P, \delta)$

In this section we revisit the problem of computing $\overrightarrow{\mathsf{H}}_\mathcal{V}(P, \delta)$ considered by Van Kreveld et al. [89]. We improve on the running time of their $O(n^4)$ time algorithm. First we thicken the input curve P by width δ. This induces a circular arc polygon $\mathcal{P}_c$, i.e. a polygon-like shape where vertices can be connected either through a straight line segment or a circular arc. $\mathcal{P}_c$ has $h = O(n^2)$ holes. We can transform this to a normal polygon $\mathcal{P}$ by replacing each circular arc with a straight line segment.

We have the following lemma:

Lemma 3.4.1. *Let $\langle p_i p_j \rangle$ be a shortcut between two vertices of P. $\langle p_i p_j \rangle$ lies fully within $\mathcal{P}_c$ if and only if it lies fully in $\mathcal{P}$.*

Proof. Let c be a circular arc of $\mathcal{P}_c$, where v is the center of the circle supporting c. Then v is a vertex of P. Let q, q' be the endpoints of c, so $\overline{qq'}$ is the line segment that is part of the boundary of $\mathcal{P}$ that replaces c. We observe that the region R bounded by c and $\langle qq' \rangle$ is convex. If R were to contain some vertex of P, then c would not be the boundary of $\mathcal{P}_c$. Furthermore, shortcuts $\langle p_i p_j \rangle$ of P have their endpoints in $\mathcal{P}_c$ and also inside of $\mathcal{P}$, so such a shortcut must intersect R zero times or twice. Since $\langle p_i p_j \rangle$ and $\langle qq' \rangle$ are both line segments, they can intersect at most once. Hence, if $\langle p_i p_j \rangle$ intersects $\langle qq' \rangle$, it must also intersect c. Hence R cannot be intersected by any shortcut $\langle p_i p_j \rangle$, which means there can be no shortcuts lying in $\mathcal{P}_c$ but not $\mathcal{P}$. □

Now all we need is to decide whether each shortcut $\langle p_i p_j \rangle$ for all $1 \le i < j \le n$ lies entirely within $\mathcal{P}$ or not. To this end, we preprocess $\mathcal{P}$ into a data structure such that for any straight line query ray ρ originated from some point inside the $\mathcal{P}$, compute the first point on the boundary of $\mathcal{P}$ hit by ρ. In other words we have a collection of h simple polygons of total complexity of $N = O(n^2)$. We use the data structure proposed by [40] of size $O(N)$ which can be constructed in time $O(N\sqrt{h} + h^{3/2}\log h + N\log N)$ and which answers queries in $O(\sqrt{h}\log N)$ time. We have $\Theta(n^2)$ shortcuts $\langle p_i p_j \rangle$ to process and need to examine whether each shortcut lies inside $\mathcal{P}$ or not i.e., whether there is a simple polygon (hole in $\mathcal{P}$) that is hit by ρ. If a shortcut lies inside $\mathcal{P}$ then we store it into the edge set of the shortcut graph proposed by Imai-Iri [72]. Otherwise we eliminate the shortcut. We originate a ray at p_i and compute the first point x on the boundary of $\mathcal{P}$ hit by the ray in $O(\sqrt{h}\log N)$ query time. All we need is to compare the length of the ray to the length of the shortcut. If $\|p_i - x\| \ge \|p_i - p_j\|$ then the shortcut lies inside $\mathcal{P}$, otherwise it does not. Once the edge set of the shortcut graph is constructed, we compute the shortest path in it. As a result we have the following theorem:

Theorem 3.4.2. *Let P be polygonal curve in $\mathbb{R}^d$ with n vertices and $\delta > 0$ be a real. One can compute $\overrightarrow{\mathsf{H}}_{\mathcal{V}}(P, \delta)$ in $O(n^3 \log n)$ time and $O(n^2)$ space.*

▸ 3.5 NP-hardness template for curve-restricted simplification

In this section we construct a template that we use to prove NP-hardness of the curve-restricted GCS problems for most of the distance measures discussed in this chapter. The template takes inspiration from the NP-hardness proofs of minimum-link path problems [88]. We believe that this template can be adapted to show hardness of other similar problems.

The template reduces from the subset sum problem. Given a set of m positive integer numbers $A = \{a_1, a_2, \ldots, a_m\}$ and an integer M, we will construct an instance of the curve-restricted GCS problem such that there exists a subset $B \subset A$ with the total sum of its integers equal to M if and only if there exists a simplified polygonal curve with at most $2m + 1$ vertices.

▸ 3.5.1 Overview

The input curve P we construct has a zig-zag pattern. It has *m split gadgets* at every other bend of the pattern, $m + 1$ *enumeration gadgets* at the other bends, and $2m$

pinhole gadgets halfway through each zig-zag segment (refer to Figure 3.7). The split and the enumeration gadgets are the same for all the distance measures, and only the pinhole gadgets vary. The construction forces any optimal simplification P' to follow a zig-zag pattern. The pinhole gadget is named as such because any segment of P' that goes through it is forced to pass through a specific point, called the *pinhole*. This limits the placements of the vertices of P'. The choice of where to place the vertex on each split gadget corresponds to the choice of including or excluding a given integer in the subset B. The x-coordinate of the vertex of P' on each enumeration gadget encodes the sum of integers in B up to that point. We will ensure that the last point of P is reachable using at most $2m + 1$ vertices only if B sums to exactly M. Different distance measures require different pinhole gadgets to make this construction work. For the reduction to be successful the following properties must hold for the construction:

1. Any segment of P' starting before a pinhole gadget and ending after the pinhole gadget must pass through the pinhole gadget's *pinhole*.

2. It must be impossible to have a segment of P' traverse multiple pinhole gadgets at once.

3. Any segment of P' where the starting vertex u is on a split or enumeration gadget, the segment goes through a pinhole, and the ending vertex v is on the next enumeration or split gadget, must have distance $\leq \delta$ to $P[u, v]$.

4. P must be polynomial in size. Specifically, only a polynomial number of polyline segments can be used and all vertices must have rational coordinates.

In Section 3.5.3 we will show that a construction with these properties implies NP-hardness.

▸ 3.5.2 Exact construction

Exact coordinates for the construction are based on two constants δ and γ. The value δ is the bound on the allowed distance between a simplification and the original curve, and γ is a constant that is significantly larger than δ. For our construction, we set $\delta = 4 \sum a_i$, and $\gamma = \delta^2$.

Split gadget The split gadgets, denoted by σ^i (for $1 \leq i \leq m$), consist of a chain of five segments (refer to Figure 3.8). Their vertex coordinates are as follows (with

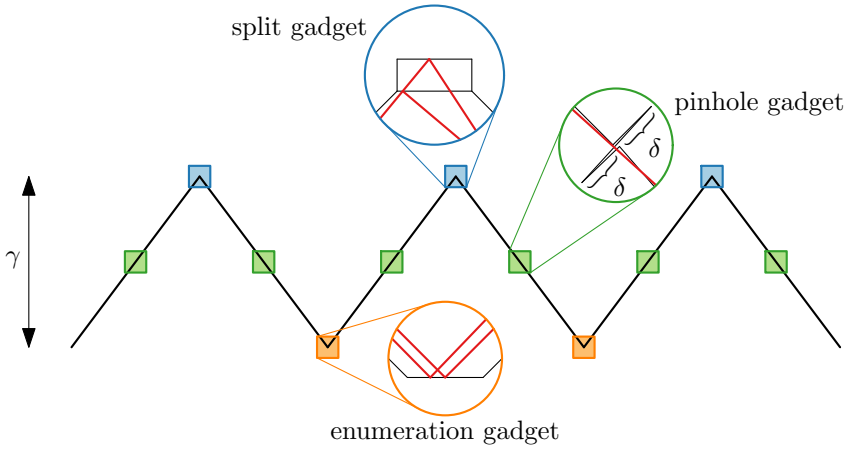


Figure 3.7: Sketch of our template curve.

vertex index denoted by subscript):

$$\sigma_1^i, \sigma_5^i = \left(\frac{3(2i-1)}{4}\gamma, \gamma\right),$$
$$\sigma_2^i, \sigma_6^i = \left(\frac{3(2i-1)}{4}\gamma + w, \gamma\right),$$
$$\sigma_3^i = \left(\frac{3(2i-1)}{4}\gamma + w, \gamma + h_i\right),$$
$$\sigma_4^i = \left(\frac{3(2i-1)}{4}\gamma, \gamma + h_i\right),$$

where $h_i = \frac{2\gamma a_i}{3\gamma - 4a_i}$ for $i \in \{1, \ldots, m\}$, and $w = \delta/2$ is the width of the split gadgets. The segments $P[\sigma_1^i, \sigma_2^i]$ and $P[\sigma_3^i, \sigma_4^i]$ are called σ^is *lower- and upper-mirror segments* respectively.

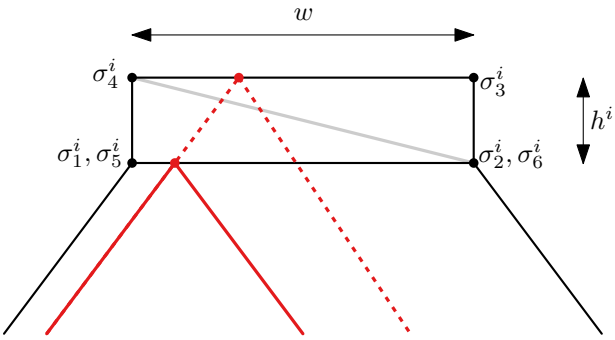


Figure 3.8: Split gadget shown in black, with a simplification in red. The vertex of the simplification can be placed on either the top or bottom horizontal (mirror) segment. The diagonal distance (in gray) is less than $\delta/\sqrt{2}$, thus all the points of the split gadget are within distance δ to either placement of the simplification vertex.

Enumeration gadget The enumeration gadgets, denoted by μ^i (for $1 \le i \le m-1$), consist of just one horizontal segment, with the following vertices:

$$\mu_1^i = \left(\frac{3i}{2}\gamma - w, 0\right),$$
$$\mu_2^i = \left(\frac{3i}{2}\gamma, 0\right).$$

There are two special enumeration gadgets: the first enumeration gadget μ^0 consists of one point with coordinates $(0,0)$ (the starting point of the curve P), and the last enumeration gadget μ^m, also called the *budget segment*, consists of two vertices with the following coordinates:

$$\mu_1^m = \left(\frac{3m}{2}\gamma - w, 0\right),$$
$$\mu_2^m = \left(\frac{3m}{2}\gamma - M, 0\right).$$

Pinhole gadget The exact structure of the pinhole gadgets depends on the curve distance measure. It is important for us however to specify the coordinates of its *pinhole*, that is, the fixed point that every simplification segment must pass through.

We denote these pinholes as c^i for $0 \le i \le 2m-1$. Their coordinates are:

$$c^i = \left(\frac{3}{8}\gamma + \frac{3i}{4}\gamma, \frac{1}{2}\gamma\right).$$

Each segment of a simplified polygonal curve directed from an enumeration to a split gadget has to pass through a pinhole with an even index, and each segment from a split to an enumeration gadget has to pass through a pinhole with an odd index.

▸ 3.5.3 Proof of the construction

As stated in the overview, there are several properties that must be satisfied for the construction to work. We will begin by showing that as long as a particular pinhole gadget inserted in the template does not significantly alter the structure of the curve, the last three properties are already satisfied by our construction regardless of the shape of the pinhole gadget. Property 2 holds because all of the pinholes have the same y-coordinate. So any segment traversing two pinholes must be a horizontal one with y-coordinate $\gamma/2$, which makes the distance between the segment and the enumeration or split gadget between the pinholes greater than δ. Property 4 obviously holds as our curve has a polynomial number of vertices with rational coordinates. For Property 3 we have the following lemma:

Lemma 3.5.1. *The split and enumeration gadgets can be covered by a vertex of a simplification on the gadget under distances* $\overrightarrow{\mathsf{H}}$, $\overleftarrow{\mathsf{H}}$, H, F, wF.

Proof. The width of an enumeration segment is $\delta/2$. Trivially, under all the distance measures, a point on it will cover the whole segment.

The width of a split gadget σ^i is $\delta/2$ as well, and the height is less than a_i which in turn is less than $\delta/2$. Then, the diagonal of the rectangle forming the gadget is less than $\delta/\sqrt{2}$ (refer for an example to Figure 3.8). Thus, any point on the split gadget is within distance δ of all the points of the gadget under any of the considered distance measures.

□

Since split and enumeration gadgets are completely covered by any points we place on them, property 3 of our construction holds as long as our pinhole gadget has distance at most δ to any line segment between a split and enumeration gadget that passes through the pinhole.

Now that we have shown our properties to hold (given a correct pinhole gadget), we will show how the construction enforces that any optimal simplification of P with at most $2m + 1$ vertices encodes a subset of A that sums to exactly M.

Lemma 3.5.2. *If the input curve P is constructed with pinhole gadgets satisfying the four template properties, any optimal simplification P' of P must have a vertex on all split and enumeration gadgets in the corresponding order, passing through the pinhole of each pinhole gadget.*

Proof. Note that we count the split gadgets starting from 1, and the enumeration segments starting from 0. We will prove that every segment of a valid simplification of size at most $2i$ of the prefix of P up to enumeration segment μ^i passes through a pinhole, and that each segment connects a split and an enumeration gadget. There are $2i$ pinhole gadgets between μ^0 and μ^i. Since P, by assumption, satisfies Property 2 (no simplification segment can traverse multiple pinholes at once), and Property 1 (any segment of a simplification starting before and ending after a pinhole gadget must pass through the pinhole), at least $2i + 1$ vertices are needed to reach μ_i from μ_0. Property 3 implies that a simplification with a vertex on each enumeration and split gadget is valid and since it has $2i + 1$ vertices it is optimal. Finding a valid simplification that does not have a vertex on each split and enumeration gadget requires additional vertices so that the simplification is still within distance δ of each gadget. This would imply that such a simplification is not optimal. □

Lemma 3.5.3. *The set of points on the ith enumeration gadget that a valid simplification curve with at most $2i + 1$ vertices can reach encodes all possible subsets of $\{a_1, \ldots, a_i\}$ in their x-coordinates: For a given subset B', the horizontal distance from the corresponding point to the right end of the enumeration segment is $\sum_{a \in B'} a$.*

Proof. From Lemma 3.5.2 we know that a simplification that reaches μ^i using at most $2i+1$ vertices has each segment pass through a pinhole. Knowing this, we can prove the lemma by induction.

Consider the base case $i = 1$. If we draw a line from μ^0 through pinhole c^0 there are two intersection points with σ^1 which are thus the possible locations of the first vertex of an optimal simplification. The coordinates of these reachable points are $(\frac{3}{4}\gamma, \gamma)$ on the lower mirror segment and $(\frac{3}{4}\gamma + \frac{3}{4}h_1, \gamma + h_1)$ on the upper mirror segment. Drawing lines from these points through the next pinhole c^1 gives two possible intersection points with μ^1: $(\frac{3}{2}\gamma, 0)$, for the line from the point on the lower mirror segment, and $(\frac{3}{2}\gamma - a_1, 0)$, for the line from the point on the upper mirror segment. The x-coordinates of these points are of the form $\frac{3}{2}\gamma - S$, where S is the

sum of the elements in any possible subset of $\{a_1\}$, namely the empty set and the set $\{a_1\}$ itself.

Now for the general case, assume that any valid simplification with $2(i-1)+1$ vertices can reach a set of points on μ^{i-1} that encodes all of the subsets of $\{a_1, \dots, a_{i-1}\}$. That is, the x-coordinate of a reachable point is of the form $\frac{3(i-1)}{2}\gamma - S$, where S is the sum of the elements in some subset of the first $i-1$ integers in A. We will now show that going from any one of these reachable points through the next two pinholes will allow us to reach precisely the points $(\frac{3i}{2}\gamma - S, 0)$ and $(\frac{3i}{2}\gamma - S - a_i, 0)$ on μ^i, corresponding to the two subsets created by either including a_i into that subset or not.

The line from $(\frac{3(i-1)}{2}\gamma - S, 0)$ through $c^{2(i-1)}$ (the pinhole between μ^{i-1} and σ^i), intersects σ^i in the point $(S - \frac{3}{4}\gamma + \frac{3}{2}\gamma i, \gamma)$ on the lower mirror line and the point $(S + \frac{2Sh_i}{\gamma} + \frac{3}{4}(h_i + \gamma(-1 + 2i)), \gamma + h_i)$ on the upper mirror line. The lines through these points and the next pinhole c^{2i-1} intersect μ^i in the points $(\frac{3i}{2}\gamma - S, 0)$ and $(\frac{3i}{2}\gamma - S - a_i, 0)$, respectively.

Thus, the set of points on the ith enumeration gadget that a valid simplification with $2i+1$ vertices can reach encodes all possible subsets of $\{a_1, \dots, a_i\}$. □

From these lemmas it follows that any point that can be reached on the last enumeration gadget using at most $2m+1$ vertices encodes a sum of the integers in a subset of A. We specifically constructed the final enumeration gadget to have its rightmost point (i.e. the endpoint of P) to be the point that encodes the value M. Thus there is a simplification curve with $2m+1$ vertices if and only if there is a subset $B \subset A$ with the total sum of its integers being M. We conclude with the following theorem.

Theorem 3.5.4. *Given a curve distance measure, if there exists a pinhole gadget that can be inserted into the described template such that the above properties hold, the curve-restricted GCS problem for that distance measure is NP-hard.*

▸ 3.5.4 NP-hardness of computing $\mathsf{F}_C(P, \delta)$, $\overleftarrow{\mathsf{H}}_C(P, \delta)$

Now, we will prove two versions of the GCS problem to be NP-hard using our template by constructing suitable pinhole gadgets.

NP-hardness of computing $\mathsf{F}_C(P, \delta)$ We construct the pinhole gadget for $\mathsf{F}_C(P, \delta)$ in the following way: It consists of a chain of five line segments that starts and ends

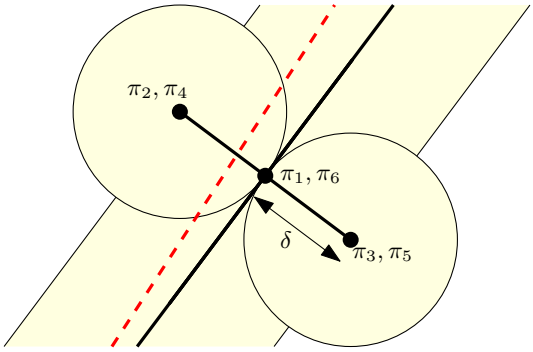


Figure 3.9: A pinhole gadget for the Fréchet distance (with even index). An invalid simplification segment is shown as a dashed line. Any valid simplification segment must allow a parametrization that visits both disks twice in alternating order. This is only possible by having the segment intersect the pinhole, the point where the disks intersect.

in the pinhole point c. The vertices of pinhole gadget π have the following coordinates, *relative to* c (refer to Figure 3.9):

$$\begin{aligned}
\pi_1, \pi_6 &= (0, 0) \ , \\
\pi_2, \pi_4 &= \left(\frac{-4\delta}{5}, \frac{3\delta}{5} \right) \ , \\
\pi_3, \pi_5 &= \left(\frac{4\delta}{5}, \frac{-3\delta}{5} \right) \ .
\end{aligned}$$

These coordinates are for pinhole gadgets with an even index. For pinhole gadgets with an odd index, the sign of the y-coordinates is inverted.

Lemma 3.5.5. *A segment traversing a pinhole gadget π, where π is constructed as described in this subsection, can only be within Fréchet distance δ of π if it intersects π's origin (the pinhole).*

Proof. The distance between the outer vertices of the pinhole gadget and the pinhole is exactly δ. This means there is a segment that passes through the origin that is within Fréchet distance δ of the pinhole gadget π. Consider a pair of parametrizations of the original and simplification, for ease of description we will use the man-dog terminology associated with the Fréchet distance. Let the man traverse P and the dog traverse P'. If we center a disk with radius δ on every vertex of P, for the

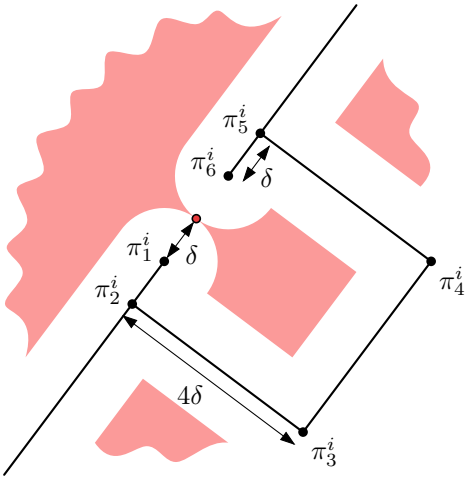


Figure 3.10: Pinhole gadget for the directed Hausdorff distance from the simplification to the original curve. The pinhole point is shown in red. The points of the area shaded in light-red have distance greater than δ to the curve. Thus no simplification segments can pass through it.

parametrizations to realize a Fréchet distance of δ the dog must pass through each disk in the order of the associated vertices. For the pinhole gadget, since the disks for π_2 and π_4 fully overlap, and the disks for π_3 and π_5 do as well, the dog must alternate between visiting these disks. Since the dog cannot walk backwards, this is only possible if the segment of P' intersects both circles in a single point. Due to how the gadget is constructed, the only point lying on both circles is the pinhole and so the simplification segment must intersect the pinhole. □

To assemble the hardness construction we translate the pinhole gadget to the middle of each of the zig-zag segments of the template curve, and combining Theorem 3.5.4 with the lemma above we obtain the following result:

Theorem 3.5.6. *Computing $\mathsf{F}_C(P, \delta)$ is NP-hard.*

NP-hardness of computing $\overleftarrow{\mathsf{H}}_C(P, \delta)$ Proving NP-hardness for computing $\overleftarrow{\mathsf{H}}_C(P, \delta)$ requires another type of pinhole gadget. Our new gadget is shown in Figure 3.10. Its vertices are as follows, *relative to c*:

For pinhole gadgets with an even index:

$$\pi_1 = \left(\frac{-3\delta}{5}, \frac{-4\delta}{5}\right), \quad \pi_2 = \left(\frac{-6\delta}{5}, \frac{-8\delta}{5}\right), \quad \pi_3 = (2\delta, -4\delta),$$
$$\pi_4 = \left(\frac{22\delta}{5}, -\frac{-4\delta}{5}\right), \quad \pi_5 = \left(\frac{6\delta}{5}, \frac{8\delta}{5}\right), \quad \pi_6 = \left(\frac{3\delta}{5}, \frac{4\delta}{5}\right).$$

For the odd-indexed pinhole gadgets this construction must be mirrored around a horizontal line through the pinhole, so the line segment $P[\pi_1, \pi_2]$ runs parallel with the line segment from the preceding split gadget to π_1.

Lemma 3.5.7. *A segment traversing a pinhole gadget π, where π is constructed as described in this subsection, can only be within directed Hausdorff distance (directed from the segment to π) δ if it intersects π's origin (the pinhole).*

Proof. π_1^i and π_6^i both have distance exactly δ to the origin so a segment that passes through it has distance δ to the gadget. Trying to pass to the right of the origin doesn't work as there is a gap of size 4δ between the gadget's segments on opposite sides of the origin. On the left side of the origin there are no segments at all that could have distance less than δ. □

To assemble the hardness construction we translate the pinhole gadget to the middle of each of the zig-zag segments of the template curve (rotating the gadget as needed as stated above). Combining Theorem 3.5.4 with the lemma above we obtain the following result:

Theorem 3.5.8. *Computing $\overrightarrow{\mathrm{H}}_C(P, \delta)$ is NP-hard.*

▸ 3.5.5 Extending the template

With our template construction, it is also possible to prove NP-hardness for the curve restricted GCS problem under the weak Fréchet, directed Hausdorff (directed from curve to simplification), and (undirected) Hausdorff distance measures. This requires new types of pinhole gadgets, but these can be difficult to design. We will therefore use an alternative approach and show how we can combine our template with the gadget introduced for Fréchet distance in Section 3.5.4 to prove NP-hardness for these distance measures, by expanding one of the properties of our construction. To see why the template and this gadget do not trivially imply NP-hardness for these measures by the steps shown earlier in this section, refer to Figure 3.11: Under distance measures like weak Fréchet, segments of P' do not have to pass exactly

through the pinhole but can also pass within a small (Euclidean) distance of the pinhole. This implies that from a single vertex on a split gadget, instead of just one point being reachable on the next enumeration gadget, there is now an interval of reachable points. For each of the points in this interval there is an interval of reachable points on the next split gadget. The union of these intervals gives a bigger interval and so the size of the intervals is increasing as they are propagated throughout the construction.

Each interval on an enumeration segment still contains the point that precisely encodes a subset sum, along with other points whose x-coordinates are within some small amount of this subset sum. If the intervals encoding the sums of different subsets stay small enough so that they never overlap with each other, the NP-hardness construction still holds. In this case the endpoint of P is contained in the interval corresponding to a subset of A with the sum of elements equal to exactly M. Therefore we can replace the first property required for our construction (Any segment of P' starting before a pinhole gadget and ending after the pinhole gadget must pass through the pinhole gadget's *pinhole*) with a slightly relaxed property below and still have proof of NP-hardness:

1. The endpoint of any segment of P' starting before a pinhole gadget and ending after the pinhole gadget must have distance less than $\frac{0.5}{2m}$ to the endpoint of the segment with the same starting point that passes exactly through the pinhole and ends on the same segment of P.

Since an interval can be propagated at most $2m$ times (once for each pinhole), the width of the interval on the final enumeration gadget is always less than 0.5 on either side of the point reached by only going through pinholes. This means the intervals will not overlap, since the points reached by only going through pinholes have distance 1 to each other. This means the construction still implies NP-hardness.

We will now show that this property holds for weak Fréchet distance, directed Hausdorff distance from curve to simplification, and undirected Hausdorff distance for the pinhole gadget depicted in Figure 3.9.

Lemma 3.5.9. *The expanded pinhole property holds for weak Fréchet distance, directed Hausdorff distance from curve to simplification, and undirected Hausdorff distance.*

Proof. For these distance measures, simplification segments from an enumeration gadget to a split gadget (and vice versa) are valid iff the segment has distance less than δ to all of the vertices of the pinhole gadget in between. As shown in the proof for Lemma 3.5.3, a segment that starts on enumeration gadget μ^{i-1} at the point $v_{\mu^{i-1}} = (\frac{3(i-1)}{2}\gamma - S, 0)$ that goes through the pinhole will intersect the split gadget

σ^i in the point $(S + \frac{3i}{2}\gamma - \frac{3}{4}\gamma, \gamma)$. So, if our intervals are to remain small enough it is impossible to have a segment from $v_{\mu^{i-1}}$ to the point $v_{\sigma^i} = (S + \frac{3i}{2}\gamma - \frac{3}{4}\gamma + \frac{0.5}{2m}, \gamma)$ with distance $\leq \delta$ to the pinhole gadget. The distance from the segment between these points to the point $\pi_2^{2(i-1)}$ (the leftmost vertex of the pinhole gadget) can be given by the equation

$$\|\langle P(v_{\mu^{i-1}}), P(v_{\sigma^i})\rangle - \pi_2^{2(i-1)}\| = \frac{\left|\frac{5\gamma+\delta(6+50\gamma m+48mS)}{m}\right|}{10\sqrt{\frac{25\gamma^2m^2+6\gamma m(1+8mS)+(1+8mS)^2}{m^2}}},$$

Given the values we chose for δ and γ in our construction ($\delta = 4\sum_{a\in A} a$, $\gamma = \delta^2$), this distance is greater than δ. The distance between the line segment starting on $v_{\mu^{i-1}}$ and ending in $(S + \frac{3i}{2}\gamma - \frac{3}{4}\gamma - \frac{0.5}{2m}, \gamma)$ to $\pi_3^{2(i-1)}$ is also greater than δ. This same argument also applies to line segments ending in the upper mirror segment, and to segments from a split gadget to an enumeration gadget. □

Combining Lemma 3.5.9 and Theorem 3.5.4 gives the following result:

Theorem 3.5.10. *Computing* $\mathsf{wF}_C(P,\delta)$, $\overleftarrow{\mathsf{H}}_C(P,\delta)$, $\mathsf{H}_C(P,\delta)$ *is NP-hard.*

In Section 3.9 (Corollary 3.9.2) we also prove strong NP-hardness for curve-restricted GCS under undirected Hausdorff distance.

▸ 3.6 Computing $\mathsf{dF}_C(P,\delta)$

In this section we present an $O(n^3)$ time algorithm for computing $\mathsf{dF}_C(P,\delta)$. Let $P = \langle p_1, \ldots, p_n\rangle$ be a polygonal curve. We first argue that there is only a discrete set of candidate points we need to consider for vertices of the output curve.

Let $\mathcal{A}$ be the arrangement of n disks of radius δ centered on the points in P, and let $C = \langle c_1, \ldots, c_m\rangle$ be the sequence of intersections between the polyline P and $\mathcal{A}$, in order of P. Since under discrete Fréchet distancethe parametrizations of the curves move in a discrete manner from vertex to vertex, the vertices of a simplification must each lie within δ of a vertex of P. No matter where in this disk a simplification vertex lies, it covers the vertex. This means that only at the points where A intersects P there is a change between which vertices of P are reached by placing a vertex of a simplification there. This leads to the following observation:

Observation 3.6.1. *Under the discrete Fréchet distance, if there exists an optimal curve-restricted simplification $P' = \langle q_1, \ldots, q_k\rangle$ of P, then there exists a subsequence of C of length k which is a simplification of P.*

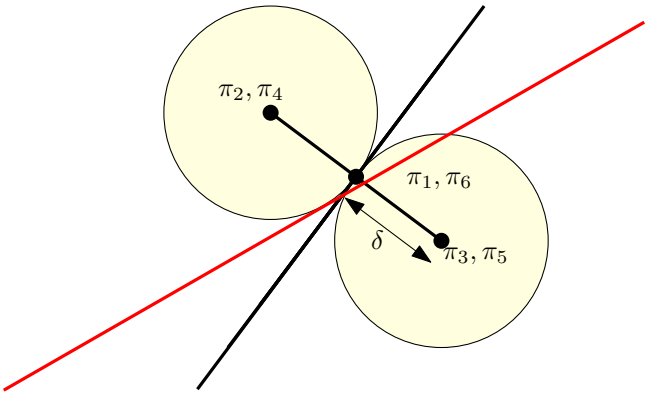


Figure 3.11: The pinhole gadget previously used for the (strong) Fréchet distance. Since weak Fréchet distance allows for non-monotonic mapping, this implies that the segments of valid simplification (example sketched in red) are not forced to pass through the pinhole. As long as a simplification segment intersects both circles, it covers the pinhole gadget.

Clearly, C consists of at most $m \in O(n^2)$ points. Bereg et al. [24] show how to compute the minimal vertex-restricted simplification of $\mathcal{A}$. We cannot apply their result directly by treating all points in C as vertices, since we do not require the simplification to be mapped to all such points, only those in P. However, we can design an algorithm in similar fashion.

Define $K(i, j)$ to be the minimum value k such that there exists a subsequence $c_1, \ldots, c_j$ of length k that has discrete Fréchet distance at most δ to the sequence $p_1, \ldots, p_i$. If no such sequence exists at all, we set $K(i, j) = \infty$ (note that this happens if and only if the distance between p_i and c_j is larger than δ). We will design a dynamic program to calculate all nm values $K(i, j)$.

First, we observe that we only need to match every point in P once. In principle, the discrete Fréchet distance is defined by a sequence of pairs of points, one from each sequence, subsequent pairs can advance in either of the two sequences. However, we never need to consider the case where a subsequent pair advances P' but does not advance P.

Observation 3.6.2. *For the optimal solution P' of length k, there exists a Fréchet matching consisting of exactly n pairs of points, each matching a unique point in P to a point in P'.*

Proof. Suppose there would be two consecutive pairs (p_i, q_j) and (p_i, q_{j+1}) matching

the same point p_i in P to different consecutive points q_j and q_{j+1} in P'. Since P' is a minimum-length subsequence of C, both q_j and q_{j+1} must also match to other points: there must be a pair (p_{i-1}, q_j) (otherwise we could eliminate the point q_j), and there must be a pair (p_{i+1}, q_{j+1}) (otherwise we could eliminate the point q_{j+1}. But then, one of the pairs (p_i, q_j) and (p_i, q_{j+1}) is superfluous: we can remove either one and still have a valid Fréchet matching between P and P'. □

This observation implies that when calculating the value of $K(i, j)$, we only have to consider values of the form $K(i-1, j')$, where $1 \le j' \le j$. Specifically, if p_{i-1} and c_j are within distance δ, then:

$$K(i,j) = \min\left(K(i-1,j), \min_{1\le j'<j}\left(K(i-1,j')+1\right)\right) \quad \text{if distance } < \delta, \text{ otherwise } \infty.$$

This definition immediately gives an $O(n^4)$ time algorithm to compute $K(n, m)$. We can improve on this by maintaining a second table with prefix minima. Let $M(i, j) = \min_{1\le j'\le j} K(i, j)$. Then we have the recursive system

$$K(i,j) = \min\left(K(i-1,j), M(i-1,j-1)+1\right) \text{ and } M(i,j) = \min\left(M(i,j-1), K(i,j)\right),$$

which can clearly be calculated in constant time per table entry, and overall saves a linear factor.

Theorem 3.6.3. *Given a polygonal curve P with n vertices and $\delta > 0$, $\mathsf{dF}_C(P, \delta)$ can be computed in $O(n^3)$ time and $O(n^2)$ space.*

It is an interesting question whether this time bound can be improved. A more greedy approach fails because of the requirement that P' is a subsequence of C, and not just a subset: a local choice to advance might have implications later.

▸ 3.7 Computing $\mathsf{F}_C(P, \delta)$ in $\mathbb{R}^1$

In this section we provide a greedy algorithm for the curve-restricted GCS problem in $\mathbb{R}^1$ under the Fréchet distance. In this version of the problem, P is one-dimensional, so all of its vertices have just a single coordinate and all fall within a single interval. For the Hausdorff distance this means P can always be simplified with at most three links: One link from the start vertex to the vertex with the smallest coordinate, one link to the vertex with the largest coordinate, and one link to the end vertex. For the Fréchet distance, however, me must still construct a curve that has a parametrization that is always close to a parametrization of P. The best way to visualize this is in

two dimensions, where we treat the time parameter of the parametrizations as the second dimension. See Figure 3.12.

We describe our algorithm using the man-dog metaphor for the Fréchet distance: Initially a man and his dog start at p_1. The dog marks it's initial position. The man walks along P. The dog only moves when it's distance to the man is exactly δ or the man has reached the end, otherwise it stays put. If the distance between the man and dog reaches δ, the dog's behavior depends on whether the man is moving in the same direction as he was the last time the dog marked its position. If the man is walking in the same direction, the dog simply follows at a distance of δ. If the man has changed direction, the dog first marks its current position before following. Once they both end the walk at p_n the dog marks its position a final time. Then we report the positions marked by the dog as P'. See Figure 3.12 and Algorithm 2. Here we present the pseudocode of the algorithm that we described. $x(.)$ indicates the x-coordinate of points.

Algorithm 2: Algorithm for curve-restricted GCS for F in $\mathbb{R}^1$.

```
1  i, j ← 1,  s ← p_i,  p'_j ← s,  j ← j + 1,  i ← i + 1
2  if x(p_i) > x(s) then Dir ← 'right'
3  else Dir ← 'left'
4  for i = 2 to n do
5      if i ≥ n then p'_j ← p_n
6      else
7          if x(p_i) ≥ x(s) then
8              if ‖p_i − s‖ ≤ δ then Continue
9              else
10                 if Dir = 'left' then p'_j ← s, s ← p_i − δ, j ← j + 1
11                 else s ← p_i − δ
12             Dir ← 'right'
13         else
14             if ‖p_i − s‖ ≤ δ then Continue
15             else
16                 if Dir = 'right' then p'_j ← s, s ← p_i + δ, j ← j + 1
17                 else s ← p_i + δ
18             Dir ← 'left'
19 return ⟨p'_1, ··· , p'_j⟩
```

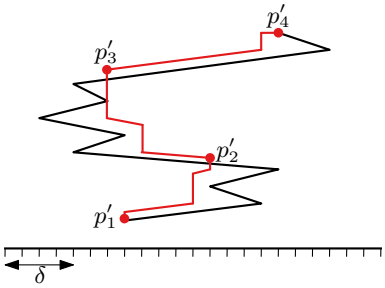


Figure 3.12: The traversal of the man of a one-dimensional curve, with the time parameter being shown as the vertical axis. The dog's movement is shown in red. The marked spots form the vertices of the simplification.

We have the following theorem:

Theorem 3.7.1. *Given a polygonal curve P in $\mathbb{R}^1$ with n vertices and $\delta > 0$, an optimal solution to the curve-restricted GCS problem under Fréchet distance can be computed in linear time.*

Proof. Let $Q = \langle q_1, \cdots, q_k \rangle$ be the optimal simplified curve and let $P' = \langle p'_1, \cdots, p'_m \rangle$ be the curve returned by the algorithm. For the sake of a contradiction, we assume that $k < m$. Observe that every time Dir changes (some turn occurs) and $\|p_i - s\| > \delta$, one vertex in P' will be added. Both curves have p_1 and p_n as their first and last vertices, respectively, in common. The remaining argument is that Q has fewer internal vertices than P' made by the two conditional loops above. The algorithm incrementally updates P' in a way that when Dir changes always a vertex will be added to P' that has the distance exactly δ to the turn vertex in P. Observe that if Q has fewer vertices it must skip one of these critical vertices which results in having distance greater than δ to turn vertices in P. Therefore $\mathsf{F}(P, Q) > \delta$ and a contradiction occurs. □

▸ 3.8 Approximation algorithm for $\mathsf{F}_{\mathcal{N}}(P, \delta)$

In this section we present an approximation algorithm for the non-restricted GCS problem under the Fréchet distance that simply discretizes the feasible space for the vertices' placement of the simplified curve. The idea is to compute a polynomial number of shortcuts in the discretized space, and (approximately) validate for each shortcut whether it is within Fréchet distance δ to a subcurve of P. For every sub-

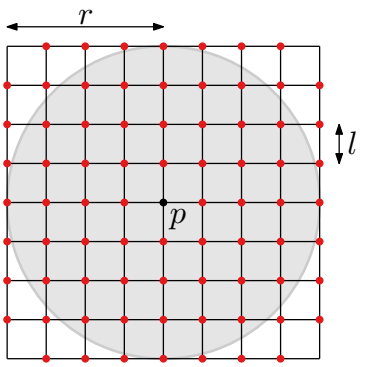


Figure 3.13: G_p as the collection of cells in $\mathsf{Prt}(\mathbb{R}^d, l)$ that intersect $B(p, r)$, and the set of corners highlighted in red.

curve of P we incrementally add the valid shortcuts to the edge set of a graph G until all the shortcuts have been processed. Once G is built, we compute the shortest path in G and return P'. To speed up the validation for each shortcut, we use a data structure to decide whether the Fréchet distance between a shortcut and a subcurve of P is at most δ.

▸ 3.8.1 The approximation algorithm

For a better understanding of our algorithm, we introduce some notation. Consider a ball $B(p, r)$ of radius $r > 0$ centered at $p \in \mathbb{R}^d$. Let $\mathsf{Prt}(\mathbb{R}^d, l)$ be a *partitioning* of $\mathbb{R}^d$ into a set of disjoint cells (hypercubes) of side length l that is induced by axis parallel hyperplanes placed consecutively at distance l. For any $1 \leq i \leq n$ we call $C_i = C_i(r, l) = \{c \in \mathsf{Prt}(\mathbb{R}^d, l) \mid c \cap B(p_i, r) \neq \emptyset\}$ a *discretization* of $B(p_i, r)$. Let $\mathcal{G}_i$ be the set of *corners* of all cells in C_i.

As we can see Algorithm 3 is a straightforward computation of valid shortcuts and a shortest path in the shortcut graph G. The `Validate` procedure takes a shortcut $\langle c_1 c_2 \rangle$ and a subcurve $P[i, j]$ as arguments and its task is to (approximately) decide whether $\mathsf{F}(\langle c_1 c_2 \rangle, P[i, j]) \leq \delta$ or not. We efficiently implement the `Validate` procedure (line 5) by means of the data structure in [50]. Let D be the value returned by the data structure in [50] that approximates $\mathsf{F}(\langle c_1 c_2 \rangle, P[i, j])$. If $D \leq (1 + \varepsilon/2)\delta$, then the `Validate` returns *true* and if $D > (1 + \varepsilon/2)\delta$ then it returns *false*.

Algorithm 3: Algorithm for non-restricted GCS for Fréchet distance.

1 **forall** $i \in \{1, \cdots, n\}$ **do** Compute $\mathcal{C}_i\big(\delta, \varepsilon\delta/(8\sqrt{d})\big)$ and $\mathcal{G}_i$;
2 $E \leftarrow \emptyset,\ V \leftarrow \emptyset,\ \mathcal{G}_1 \leftarrow p_1 \cup \mathcal{G}_1,\ \mathcal{G}_n \leftarrow p_n \cup \mathcal{G}_n$;
3 **forall** C_i **and** C_j, **with** $1 \le i \le j \le n$ **do**
4 **forall** $c_1 \in C_i$ **and** $c_2 \in C_j$ **do**
5 **if** `Validate`$(\langle c_1 c_2\rangle, P[i,j]) = true$ **then**
 $E \leftarrow E \cup \langle c_1 c_2\rangle,\ V \leftarrow V \cup \{c_1, c_2\}$;
6 **return** the shortest path between p_1 and p_n in $G = (V, E)$.

▸ 3.8.2 Proof of correctness and bounds

Here we slightly rephrase the theorem that refers to the data structure in [50] according to our terminology:

Lemma 3.8.1 (Theorem 5.9 in [50])**.** *Let P be a polygonal curve in $\mathbb{R}^d$ with n vertices and let $0 < \varepsilon \le 1/8$ be a real value. One can construct a data structure of size $O\big((\varepsilon^{-d}\log^2(1/\varepsilon))n\big)$ and construction time of $O\big((\varepsilon^{-d}\log^2(1/\varepsilon))n\log^2 n\big)$, such that for any query segment $\langle ab\rangle$ in $\mathbb{R}^d$ and two vertices p_i and p_j in P with $1 \le i \le j \le n$, one can compute a $(1+\varepsilon)$-approximation of $\mathsf{F}(\langle ab\rangle, P[i,j])$ in $O(\varepsilon^{-2}\log n \log\log n)$ query time.*

Lemma 3.8.2. *Let $0 < \varepsilon \le 1$ and let $\langle ab\rangle$ be a segment in $\mathbb{R}^d$ such that a and b are confined within some two cells $h' \in C_i$ and $h'' \in C_j$, respectively, with $1 \le i \le j \le n$. If $\mathsf{F}(\langle ab\rangle, P[i,j]) \le \delta$, then for all corners $c' \in h'$ and $c'' \in h''$ `Validate`$(\langle c'c''\rangle, P[i,j])$ returns* true.

Proof. Let c' be an arbitrary corner of h' and c'' an arbitrary corner of h''. Note that $\mathsf{Diam}(h') = \mathsf{Diam}(h'') = \sqrt{d}\cdot(\varepsilon\delta/8\sqrt{d}) = \varepsilon\delta/8$, where $\mathsf{Diam}(h')$ and $\mathsf{Diam}(h'')$ are the diameters of cells h' and h'', respectively. Hence $\ell_1 = \|a - c'\| \le (\varepsilon/8)\delta$ and $\ell_2 = \|b - c''\| \le (\varepsilon/8)\delta$. Given the two segments $\langle ab\rangle$ and $\langle c'c''\rangle$ the Fréchet distance between them is $\mathsf{F}(\langle c'c''\rangle, \langle ab\rangle) = \max\{\ell_1, \ell_2\} \le (\varepsilon/8)\delta$ by [16]. We build the data structure of Lemma 3.8.1 for `Validate` with respect to parameter $\varepsilon/8$ and the whole curve P that leads to a $(1+\varepsilon/8)$-approximation of the Fréchet distance between $\langle c'c''\rangle$ and $P[i,j]$ returned by `Validate`$(\langle c'c''\rangle, P[i,j])$. Since $D \le (1+\varepsilon/8)\cdot\mathsf{F}(\langle cc'\rangle, P[i,j])$ by applying a triangle inequality between the segments and path $P[i,j]$ we have:

$$\frac{D}{(1+\varepsilon/8)} \le \mathsf{F}(\langle c'c''\rangle, P[i,j]) \le \mathsf{F}(\langle ab\rangle, P[i,j]) + \mathsf{F}(\langle c'c''\rangle, \langle ab\rangle) \le \delta + \varepsilon\delta/8 = (1+\varepsilon/8)\delta.$$

Therefore $D \leq (1 + \varepsilon/8)^2\delta = 1 + \varepsilon/4 + \varepsilon^2/64 < (1 + \varepsilon/2)\delta$, for any $0 < \varepsilon < 1$, and $\texttt{Validate}(\langle c'c'' \rangle, P[i, j])$ returns *true*.

□

Lemma 3.8.3. *There exists a $P' = \mathsf{F}_{\mathcal{N}}(P, \delta)$ such that every link $\langle ab \rangle$ in P' does not match to a proper subsegment of $P[i, i + 1]$, i.e., $P(i, i + 1)$ for all $1 < i < n$.*

Proof. We use proof by contradiction. Assume that such a P' fulfilling the condition above does not exist, hence there is a link $\langle ab \rangle$ whose endpoints match to neither p_i nor p_{i+1}, for some $1 < i < n$. Let (σ, θ) be a Fréchet matching realizing $\mathsf{F}(P, P') \leq \delta$ and let t_i and t_{i+1} be two real values with $0 \leq t_i < t_{i+1} \leq 1$ such that $\sigma(t_i) = i$, $\sigma(t_{i+1}) = i + 1$ and $\theta(t_i) = x$, $\theta(t_{i+1}) = y$, for some $1 < x < y < \#P'$. Note that $\langle ab \rangle$ matches to neither p_i nor p_{i+1}, thus $x < a < b < y$. Now let $P'' = P'[1, x] \circ \langle xy \rangle \circ P'[y, n]$ where $P \circ Q$ denotes the concatenation of two curves P and Q where the ending point of P is the starting point of Q. Clearly, $\mathsf{F}(P, P'') \leq \delta$ due to $\mathsf{F}(P[i, i + 1], \langle xy \rangle) \leq \delta$ and also P'' has the same number of vertices as P' has (see Figure 3.14). Therefore, $P'' = \mathsf{F}_{\mathcal{N}}(P, \delta)$ exists and this is a contradiction. □

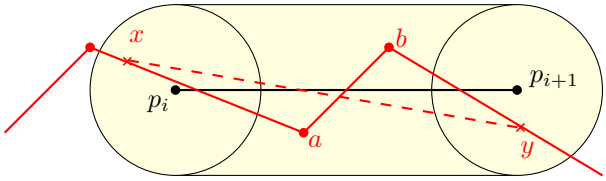


Figure 3.14: The shortcut $\langle xy \rangle$ has Fréchet distance at most δ to $P[i, i + 1]$.

Lemma 3.8.4. *Let $\langle ab \rangle$ be an arbitrary link in $P' = \mathsf{F}_{\mathcal{N}}(P, \delta)$ fulfilling the condition in Lemma 3.8.3. The following statements hold:*

1. *There exist a sublink $\langle a'b' \rangle \subseteq \langle ab \rangle$ and an integer pair (i, j) with $1 \leq i \leq j \leq n$ such that $\mathsf{F}(\langle a'b' \rangle, P[i, j]) \leq \delta$.*

2. *There exist an integer pair of cells (h', h'') and an integer pair (i, j) with $1 \leq i \leq j \leq n$ and $h' \in C_i$, $h'' \in C_j$, such that a' and b' are confined within h' and h'', respectively and $\texttt{Validate}(\langle c'c'' \rangle, P[i, j])$ returns* true *for all corners $c' \in h'$ and $c'' \in h''$.*

Proof. (1) Let P' be a solution to $\mathsf{F}_{\mathcal{N}}(P, \delta)$ satisfying Lemma 3.8.3. Then $\langle ab \rangle$ as an arbitrary link in P' cannot match to a proper subsegment in $P[i, i+1]$ for all $1 < i < n$. Now let (σ, θ) be a Fréchet matching realizing $\mathsf{F}(P, P') \leq \delta$ and let t_a and t_b be two

real values with $0 \leq t_a < t_b \leq 1$ such that $\sigma(t_a) = p$, $\sigma(t_b) = q$ and $\theta(t_a) = a$, $\theta(t_b) = b$. We then have two cases:

- $\langle ab \rangle$ only intersects $B(p_i, \delta)$. Now let $0 \leq t_i < 1$ be a value such that $\sigma(t_i) = i$. Then clearly, $\theta(t_a) \leq \theta(t_i) \leq \theta(t_b)$ and correspondingly, $\sigma(t_a) \leq \sigma(t_i) = i \leq \sigma(t_b)$. Hence, it holds that $1 \leq p \leq i \leq q \leq n$ for some $1 < i < n$. Therefore there exists a sublink $\langle a'b' \rangle = B(p_i, \delta) \cap \langle ab \rangle$ and an integer pair (i, j) with $i = j$ such that $\mathsf{F}(\langle a'b' \rangle, P[i, j]) \leq \delta$ (see Figure 3.15 (a)).

- $\langle ab \rangle$ intersects all the balls $B(p_i, \delta), \cdots, B(p_j, \delta)$ for some $1 \leq i < j \leq n$ in order from i to j. Now let $0 \leq t_i < t_j \leq 1$ be a value such that $\sigma(t_i) = i$ and $\sigma(t_j) = j$. Then clearly, $\theta(t_a) \leq \theta(t_i) < \theta(t_j) \leq \theta(t_b)$ and correspondingly, $\sigma(t_a) \leq \sigma(t_i) < \sigma(t_j) \leq \sigma(t_b)$. Hence, it holds that $1 \leq p \leq i < j \leq q \leq n$ for some $1 \leq i < j \leq n$. Now let $\langle a'b' \rangle = P'[\theta(t_i), \theta(t_j)]$. Following the Fréchet matching (σ, θ), we have $\mathsf{F}(\langle a'b' \rangle, P[i, j]) \leq \delta$ (see Figure 3.15 (b)).

Therefore such $\langle a'b' \rangle \subseteq \langle ab \rangle$ and pair (i, j) with $1 \leq i \leq j \leq n$ exist.

(2) By Property 1, we know that $\langle a'b' \rangle$ and an integer pair (p_i, p_j) with $1 \leq i \leq j \leq n$ exist such that $\mathsf{F}(\langle a'b' \rangle, P[i, j]) \leq \delta$. Plugging this into Lemma 3.8.2 completes the proof. □

The following lemmas allow us to conclude Theorem 3.8.8.

Lemma 3.8.5. *The shortest path P'_{alg} returned by Algorithm 3 exists and $\mathsf{F}(P, P'_{\mathsf{alg}}) \leq (1 + \varepsilon)\delta$.*

Proof. Let $P' = \mathsf{F}_{\mathcal{N}}(P, \delta)$, and $\langle ab \rangle$ and $\langle bc \rangle$ be two consecutive links in P'. By Lemma 3.8.4 (Property 2) there exist an integer pair (i, j) with $1 \leq i \leq j \leq n$, and two corners $c_i \in h \in C_i$ and $c_j \in h' \in C_j$ close to a sublink $\langle xw \rangle$ of $\langle ab \rangle$ such that $\texttt{Validate}(\langle c_i c_j \rangle, P[i, j])$ returns *true*. We have two following cases:

- $b \in B(p_j, \delta)$: Then there exists an integer k with $1 \leq j \leq k \leq n$, and two corners c'_j and c_k close to a sublink $\langle yz \rangle$ of $\langle bc \rangle$ such that $\texttt{Validate}(\langle c'_j c_k \rangle, P[j, k])$ returns *true*.
  Since $P'[x, y]$ has only one intermediate vertex i.e., b that lies within $B(p_j, \delta)$, it is easy to see that $\mathsf{F}(\langle xy \rangle, p_j) \leq \delta$ and therefore, $\texttt{Validate}(\langle c_j c'_j \rangle, p_j)$ returns *true* as well.

- $b \notin B(p_j, \delta)$: Then there must exist an integer k such that $1 \leq j + 1 \leq k \leq n$, and two corners c_{j+1} and c_k close to a sublink $\langle yz \rangle$ of $\langle bc \rangle$ such that the procedure $\texttt{Validate}(\langle c_{j+1} c_k \rangle, P[j + 1, k])$ returns *true*. Now again $P'[x, y]$ has only one

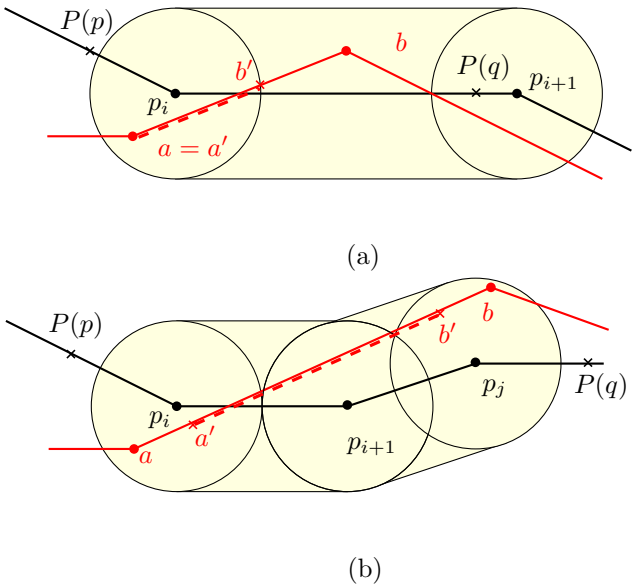


Figure 3.15: (a) Any point on the subsegment $\langle a'b' \rangle$ can be matched to p_i, thus $\mathsf{F}(\langle a'b' \rangle, P[i,j]) \leq \delta$ with $i = j$. (b) Following the optimal matching, $\mathsf{F}(\langle a'b' \rangle, P[i,j]) \leq \delta$ with $i < j$.

intermediate vertex, b, that lies within $C(\langle p_j p_{j+1} \rangle, \delta)$, where $C(Q, r)$ denotes the cylinder of width r around the segment Q. Thus $\mathsf{F}(\langle xy \rangle, P[j, j+1]) \leq \delta$ and therefore, `Validate`$(\langle c_j c_{j+1} \rangle, P[j, j+1])$ returns *true*.

This indicates that for any two adjacent links in P' we have three links in the shortcut graph. Since P' is connected the graph has a connected path P'_{alg}. Note that if such a path, P'_{alg}, is found by the algorithm is not the shortest path in G, then there must be another path which leads to the existence of a shortest path either way. The proof for a single link in P' is trivial. To prove $\mathsf{F}(P, P'_{\mathsf{alg}}) \leq (1+\varepsilon)\delta$, let e be a link in P'_{alg} and P_e be the corresponding subcurve in P such that `Validate`(e, P_e) has returned *true*. This implies that $D \leq (1+\varepsilon/2)\delta < (1+\varepsilon)\delta$ as well thus $\mathsf{F}(e, P_e) \leq (1+\varepsilon)\delta$, therefore $\mathsf{F}(P, P'_{\mathsf{alg}}) = \max_{e \in P'_{\mathsf{alg}}} \mathsf{F}(e, P_e) \leq (1+\varepsilon)\delta$. □

Lemma 3.8.6. *Let $P' = \mathsf{F}_{\mathcal{N}}(P, \delta)$ and let P'_{alg} be the curve returned by Algorithm 3. Then $\#P'_{\mathsf{alg}} \leq 2(\#P' - 1)$.*

Proof. Let $m = \#P'$ where $P' = \mathsf{F}_{\mathcal{N}}(P, \delta)$. Now for all $m - 2$ ($m > 2$) intermediate links in P' i.e., $P[2, m]$, the number of links in $P'_{\mathsf{alg}}[c_2, c_m]$ where $c_1 \in B(p_2, \delta)$ and $c_m \in B(q_m, \delta)$ is at most $2(m-2)+1$ in the worst case. This together with the two first and last links $\langle c_1 c_2 \rangle$ and $\langle c_m c_{m+1} \rangle$ in P'_{alg}, respectively, results in $\#P'_{\mathsf{alg}} \leq 2m - 1$. □

Lemma 3.8.7. *Algorithm 3 uses* $O\Big(\varepsilon^{-d} n \log n\big(\log^2(1/\varepsilon)\log n + \varepsilon^{-(d+2)} n \log\log n\big)\Big)$ *time and* $O\big((\varepsilon^{-d}\log^2(1/\varepsilon))n\big)$ *space.*

Proof. The number of cells in each ball is bounded by $O\big((\delta/(\varepsilon\delta/8\sqrt{d}))^d\big) = O(\varepsilon^{-d})$. There are $O(n^2)$ pairs of balls, hence, we have $O(\varepsilon^{-2d}n^2)$ pairs of corners c' and c'' to pass to the `Validate` procedure in order to determine whether $\mathsf{F}(\langle c'c'' \rangle, P[i, j]) \leq (1 + \varepsilon)\delta$ or not for all $c' \in C_i$ and $c'' \in C_j$ with $1 \leq i \leq j \leq n$. On the other hand by Lemma 3.8.1 we speed up the `Validate` procedure. The construction time is $O\big((\varepsilon^{-d}\log^2(1/\varepsilon))n\log^2 n\big)$ and its query takes $O(\varepsilon^{-2}\log\log\log n)$ time per $\langle c'c'' \rangle$. Because we have $O(\varepsilon^{-2d}n^2)$ calls on the `Validate` procedure, the whole validation process takes $O((\varepsilon^{-2d+2}n^2 \log n \log\log n))$ time. To get the total runtime we must add the construction time of the data structure in Lemma 3.8.1 so we get $O\Big(\varepsilon^{-d} n \log n\big(\log^2(1/\varepsilon)\log n + \varepsilon^{-(d+2)} n \log\log n\big)\Big)$ total time as claimed. The space follows from Lemma 3.8.1. □

Theorem 3.8.8. *Let P be a polygonal curve with n vertices in $\mathbb{R}^d$, $\delta > 0$, and let $P' = \mathsf{F}_{\mathcal{N}}(P, \delta)$. For any $0 < \varepsilon \leq 1$, one can compute in $O^*(n^2 \log n \log\log n)$ time and $O^*(n)$ space a non-restricted simplification P^* of P such that $\#P^* \leq 2(\#P' - 1)$ and $\mathsf{F}(P, P^*) \leq (1 + \varepsilon)\delta$. Here, O^* hides factors polynomial in $1/\varepsilon$.*

Corollary 3.8.9. *Theorem 3.8.8 also holds for* $\mathsf{wF}_{\mathcal{N}}(P, \delta)$.

Proof. One can modify the data structure in Lemma 3.8.1 to support queries under the weak Fréchet distance by regarding the weak Fréchet distance instead of the strong Fréchet distance in preprocessing stage (see [50] for more details on their data structure). Also following the fact that the triangle inequality holds for the weak Fréchet distance, Lemma 3.8.2, Lemma 3.8.3 and Lemma 3.8.4 work when applying the weak Fréchet distance between a link and a subcurve. Therefore, Corollary 3.8.9 follows from the aforementioned lemmas. □

▸ 3.9 Strong NP-hardness for computing $\mathsf{H}_{\mathcal{N}}(P, \delta)$

We show that all versions of the GCS problem under the undirected Hausdorff Distance are strongly NP-hard. This was shown for the vertex-restricted case by Van

Kreveld et al. [89] by a reduction from Hamiltonian cycle in segment intersection graphs. Their proof mostly extends straightforwardly to the curve-restricted and unrestricted case; however, because of the increased freedom in vertex placement we must take some care in the exact embedding of the segment graph: for instance, segments that intersect at arbitrarily small angles could potentially cause the reduction to produce coordinates with unbounded bit complexity. For this reason, we here reduce from a more restricted class of graphs: *orthogonal* segment intersection graphs. For completeness, we present the full adapted proof.

Czyzowicz et al. [47] show that Hamiltonian cycle remains NP-complete in 2-connected cubic bipartite planar graphs, and Akiyama et al. [14] prove that every bipartite planar graph has a representation as an intersection graph of orthogonal line segments. Hence, Hamiltonian cycle in orthogonal segment intersection graphs is NP-complete.

Let S be a set of n horizontal or vertical line segments in the plane. We may assume the segment endpoints have integer coordinates that are linear in n and that all intersections are proper intersections. We further assume that the intersection graph of S is connected (if not, it will not have a Hamiltonian cycle). Set $\delta = \frac{1}{8}$, and let $D \subset \mathcal{R}^2$ be the Minkowski sum of S and a closed ball of radius δ; that is, D is the set of all points at distance at most δ from S.

Let P be an initial polyline that stays on the union of S, and covers all segments of S many times. For instance, we may begin with a spanning tree of the intersection graph of S, traverse the tree using a depth-first search, and whenever we encounter a new segment we visit both endpoints before leaving through the intersection point with the next segment. Such a path has linear complexity. Then, we make a linear number of copies of this path and concatenate them, creating a path P of $O(n^2)$ complexity. Now, P has every ordered permutation of the segments as a subsequence of its vertices, allowing us to freely walk over the arrangement of S when creating an output polyline. We also add some additional links to P so that it starts and ends at the same point.

An output polyline P' with Hausdorff distance at most δ to P must in particular visit the δ-disks around all endpoints of S, while staying inside D. Since no two such disks are visible to each other within D unless they are endpoints of the same segment (see Figure 3.16), we will need at least one more vertex in P' for each segment, in order to switch to the next. Any output trajectory P' will need to have at least $3n + 1$ vertices (the last vertex is the same as the starting vertex).

Clearly, if the intersection graph of S has a Hamiltonian cycle, following this cycle will yield a solution using $3n + 1$ vertices by simply placing a vertex at each segment intersection and two intermediate vertices at each segment endpoint (see

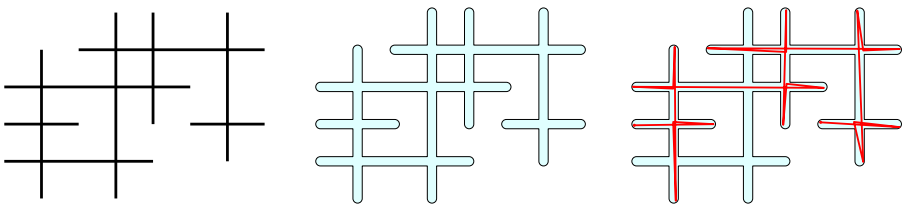


Figure 3.16: The construction: (a) the segments S, (b) the region D, (c) a partial simplification.

Figure 3.16). On the other hand, any solution that uses only $3n + 1$ vertices must be of this form: if we visit any segment more than once (and place a vertex there) we must place at least four vertices on (close to) such a segment. Theorem 3.9.1 now follows.

Theorem 3.9.1. *The non-restricted GCS problem under undirected Hausdorff distance is strongly NP-hard.*

Since a solution to the reduction never benefits from placing vertices not on P, we also immediately obtain an improvement for the special case $\mathsf{H}_C(P, \delta)$.

Corollary 3.9.2. *The curve-restricted GCS problem under undirected Hausdorff distance is strongly NP-hard.*

▸ 3.10 Strong NP-hardness for computing $\overleftarrow{\mathsf{H}}_{\mathcal{N}}(P, \delta)$

We will prove that the non-restricted GCS problem using directed Hausdorff distance in the direction $P \longrightarrow P'$ is NP-hard. We use several steps. The key difficulty in solving GCS for this distance measure lies in a problem we dub Segment Polyline Cover: Given a set L of line segments in the plane and integer k, find a polyline P such that every segment in L is *covered* by P (all points on the segment are contained in least one segment of P), and P has at most k links.

Our approach is to show that the Segment Polyline Cover problem is hard by a reduction from the Hamiltonian Path problem on ray intersection graphs, and then reduce this problem to GCS. The first reduction is based on the following idea.

Lemma 3.10.1. *Let G be a ray intersection graph with n vertices. There exists a set L of $2n$ segments such that G has a Hamiltonian cycle if and only if there is a polygon covering L with $2n$ vertices.*

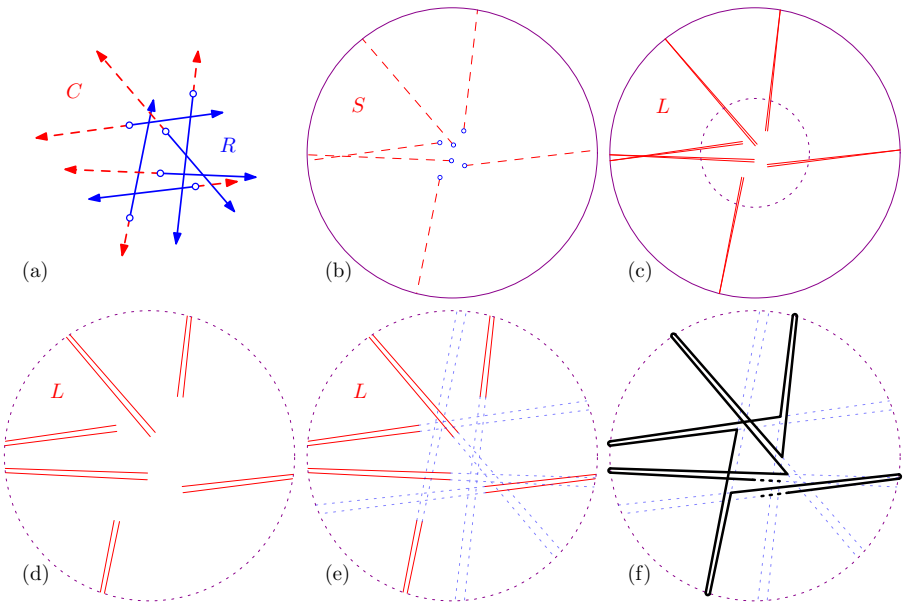


Figure 3.17: The idea for a small example (which does not admit a Hamiltonian cycle). (a) A set of rays R (blue) whose intersection graph is G, and the complement C (red, dashed). (b) Zooming out until we can draw a circle that contains all intersections among rays in C. (c) Replacing each ray in C by a *needle*. (d) Zooming back in. (e) The extensions of the needles (blue, dotted) correspond to the original rays. (f) A polygon covering all needles must correspond to a Hamiltonian cycle in G (here, there is no solution).

We sketch the proof of Lemma 3.10.1 first. The high level proof idea is illustrated in Figure 3.17. Let R be a set of rays in $\mathbb{R}^2$, and G its intersection graph. The *complement* of a ray r is the ray with the same origin and the same supporting line as r which points in the opposite direction. Let C be the complement of R. We cut the rays in C to a set of segments S in such a way that C and S have the same intersection graph. Then we replace each segment $s \in S$ by a *needle*: a pair of segments both very close to s that share one endpoint (different from the corresponding ray's origin). Let L be the resulting set of $2n$ segments. Now, any polygon with $2n$ segments covering L must use the two edges of one needle consecutively (since, by construction, the extension of these segments does not intersect the supporting line of any other segment), and it can connect an edge from one needle to an edge of another needle

exactly when the corresponding original rays in R intersect.

Though the idea is conceptually simple, there are several difficulties in turning Lemma 3.10.1 into a proof that GCS is NP-hard.

- The simple idea above is phrased in terms of a Hamiltonian cycle and covering segments by a polygon; for our proof we need to use a polyline. We need to be careful in how to handle the endpoints.
- We need to establish that the Hamiltonian Path problem is indeed NP-hard on ray intersection graphs.
- We need to specify how to embed a ray intersection graph as an actual set of rays with limited bit complexity.
- We need to model the input to GCS as an instance of the Segment Polyline Cover problem. Specifically, the complement of a set of rays is not necessarily connected; but the input to GCS must be connected.
- The Segment Polyline Cover problem closely resembles GCS for $\delta = 0$; to extend it to the case $\delta > 0$ we (again) need to carefully consider the complexity of the embedding.

Most of these challenges can be overcome, as we show in the remainder of this section. However, since the problem of recognizing if a graph can be embedded as a set of intersecting rays is complete for the existential theory of the reals [37], we know that there are ray intersection graphs that cannot be embedded by a set of rays with subexponential bit complexity, unless **NP** = $\exists\mathbb{R}$. We work around this problem by considering a smaller class of graphs, and allowing a superpolynomial grid for our embeddings, which we show is sufficient to prove the given variant of GCS is NP-hard .

▸ 3.10.1 Hamiltonian cycle on ray intersection graphs

We will show that each circle graph can be embedded as a ray intersection graph with polynomial bit complexity. To show this, we construct a set of n points that lie on a convex, increasing curve such that all chords connecting a pair of points can be extended to a ray to the right, and none of these rays will intersect below the curve. This requires the curve to grow very fast. We use the points $(x, x!)$ for $x \in [1..n]$, where $x! = \Pi_{i=1}^{x} i$ is the factorial function. Indeed, these points have the following property.

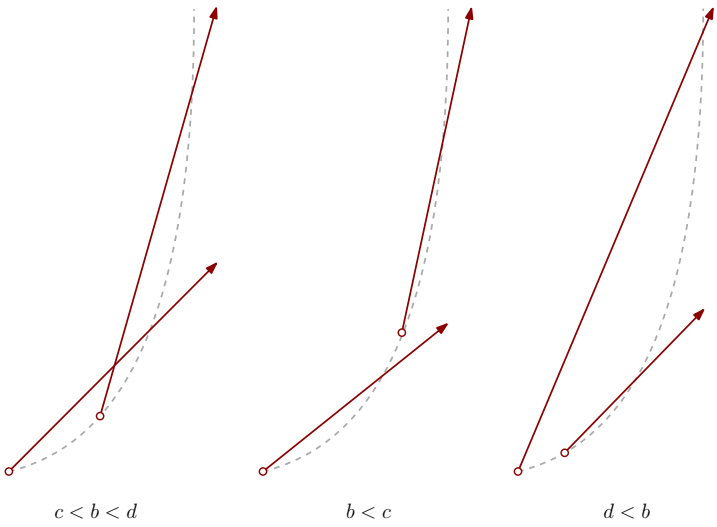


Figure 3.18: The three cases for two rays. a and b are the x-coordinates of the points where the first ray intersects with the curve $y = x!$, and c and d are those values for the second ray. Regardless of the case, the rays will not intersect below the curve.

Lemma 3.10.2. *Let $a, b, c, d \in [1..n]$ be four numbers such that $a < b$ and $c < d$. Let A be the ray starting at $(a, a!)$ and containing $(b, b!)$, and let B be the ray starting at $(b, b!)$ such that $B \subset A$. Similarly, let C be the ray starting at $(c, c!)$ and containing $(d, d!)$, and let D be the ray starting at $(d, d!)$ such that $D \subset C$. Then B and D do not intersect; hence, A and C intersect if and only if $A \setminus B$ and $C \setminus D$ intersect.*

Proof. Since every ray is drawn between two points on the curve of the function $x!$, we know that it intersects this curve only at these points. The distance between y-coordinates of successive points keeps rapidly increasing as x increases, but the distance between x-coordinates of successive points is constant. Thus the slope of a ray r_1 whose intersection points with the curve lie to the right of those of ray r_2 will be greater than the slope of r_2. Without loss of generality we assume $a < c$. There are three possible cases, see Figure 3.18:

- $c < b < d$: Here it is clear that A and C will intersect at an x-coordinate somewhere between c and b, and so B and D will not intersect.
- $b < c$: Here we can easily see that B and D do not intersect, as B starts below D and has a lower slope.

- $d < b$: Whereas the first two cases only require the curve to be convex and increasing, this case also requires the function to grow quick enough: Since D starts to the left of B it could possibly intersect B if its slope was higher. We will now show, however, that the factorial function grows quick enough so that this cannot happen. For a fixed b, the lowest slope that B can have is when $a = 1$. The highest slope that D can have occurs when $c = b - 2$ and $d = b - 1$. The slope of B is equal to the slope of A, which would be $\frac{b!-1!}{b-1} = b \cdot (b-2)! - \frac{1}{b-1}$. The slope of D (and C) in this scenario would be $\frac{(b-1)!-(b-2)!}{1} = (b-2) \cdot (b-2)!$. We can see that the slope of B is higher than D if $2 \cdot (b-2)! \geq \frac{1}{b-1}$ which obviously holds for all $b > 2$. So since B starts above D and has higher slope, B and D will not intersect.

□

Once we have constructed these points we can "unroll" any circle graph by picking one chord endpoint on the circle to be the first point and then traversing the circle in clockwise order and assigning each chord endpoint we encounter the next point of our set. See Figure 3.19 for a sketch. Because the y-coordinate for a point will not grow bigger than $O(n^n)$ we can represent the points using polynomial bit complexity. At this point, we have shown that circle graphs are contained in ray intersection graphs. In fact, our construction gives a bit more:

Theorem 3.10.3. *The class of circle graphs is contained in the class of ray intersection graphs. Furthermore, every circle graph can be embedded as the intersection graph of a set of rays such that:*

- *every ray is grounded on a common curve* (grounded ray intersection graph [37])*;*
- *every ray points towards the upper right quadrant* (downward ray intersection graph [37])*;*
- *every ray is described by a point and a vector with polynomial bit complexity.*

Next, we show that the Hamiltonian path problem is NP-hard on ray intersection graphs, and in particular, on ray intersection graphs with polynomial bit complexity.

We reduce from the Hamiltonian path problem on circle graphs. We make use of the proof from Damaschke [48]. He shows that the Hamiltonian cycle problem is NP-hard on circle graphs, by reducing from Hamiltonian cycle in cubic bipartite graphs. He also claims that there is an easy adaptation that shows the Hamiltonian path problem is also NP-hard for circle graphs. We will start by making this adaptation explicit: We construct an instance of the circle graph problem as described in [48],

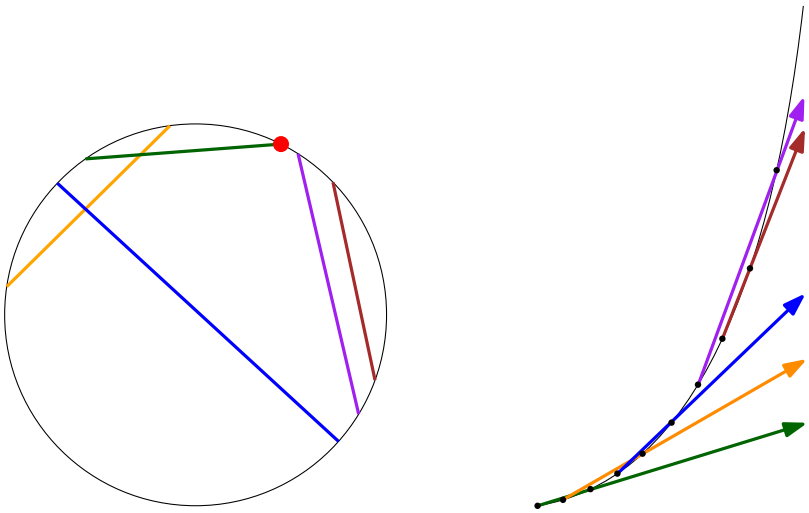


Figure 3.19: (a) A circle graph with colors assigned to the chords. The chosen starting point is marked with a red dot. (b) Unrolled version of (a), by assigning chord endpoints in counterclockwise order to points on the convex curve they can be extended into rays without intersecting.

but then we replace one of the X-chords with two parallel chords close to where the X-chord was, so that they both intersect the same chords that were intersected by the X-chord. For both of the new chords we then add one new chord that only intersects that chord and no others. Now we know that the circle graph will have a Hamiltonian path if and only if the bipartite graph has a Hamiltonian cycle. From Theorem 3.10.3 we now immediately have:

Corollary 3.10.4. *The Hamiltonian path problem is NP-hard on intersection graphs of rays that have a polynomial bit complexity.*

▸ 3.10.2 Connected segment polyline cover

Next, we introduce the connected segment polyline cover problem, and show that it is NP-hard by a reduction from the Hamiltonian path problem on circle graphs through the construction outlined above.

Connected segment polyline cover is a problem where we are given a set L of n

line segments whose union is connected, and an integer k. We must decide if there exists a polyline of k links that fully covers all segments in L.

We start by embedding the circle graph as a ray intersection graph in the manner outlined above. Then, we compute all intersection points between supporting lines of the rays. One of these intersection points will have the lowest y-coordinate. We will then choose a value that is lower than this lowest y-coordinate, which we will denote as y_ℓ. For each ray r, let p_r be its starting point. Let $\bar{r}$ be r's complement: the part of the supporting line that is not covered by r. Let $\tilde{r}$ be the part of $\bar{r}$ that has $y \geq y_\ell$. Now we construct a *needle* for each ray's complement: Two line segments that share one endpoint at the point where $\bar{r}$ has y-coordinate y_ℓ. The other endpoint for both segments lies very close to p_r. The endpoints are on opposite sides of the ray starting point so we get a wedge-like shape that runs nearly parallel to $\tilde{r}$. In addition to these $2n$ segments, which we will refer to as *needle segments*, we create three more segments which we will refer to as the *leading segments*: We create one horizontal segment we call s_h with y-coordinate between y_ℓ and the lowest intersection point between ray supporting lines, that starts far to the right of the needle segments and ends to the left of them, intersecting all of the needles. Attached to s_h is a large vertical segment we call s_v, running up to a point above the highest starting point of a ray. Attached to that is another horizontal segment we call s_t, this one being short and ending to the left of any ray starting point. See Figure 3.20 for a sketch.

Now we have $2n + 3$ segments in total, where n is the number of chords in the original graph.

Lemma 3.10.5. *We can cover all segments using a polyline of $2n + 3$ links if and only if the circle graph has a Hamiltonian path.*

Proof. Consider that since none of the segments are collinear and no three segments intersect in the same point, a suitable polyline must fully cover one segment with each link. For a polyline to be able to bend from fully covering one segment to fully covering another, either the segments must have a shared endpoint, or the supporting lines of the segments must intersect in a point not contained in either segment. Segment s_h intersects all needle segments in their interior and is parallel to s_t, so we know that a suitable polyline must start[3] by covering s_h, and it must bend at the common endpoint with s_v and then fully cover s_v. All of the intersection points between s_v and the supporting lines of needle segments lie below the endpoint it shares with s_h, so to be able to cover s_v with the second link the polyline must next connect to s_t, meaning it bends at the shared endpoint of s_v and s_t. The third link is

[3] A suitable polyline could also end with s_h, but we will define the polyline to be in this direction for ease of notation

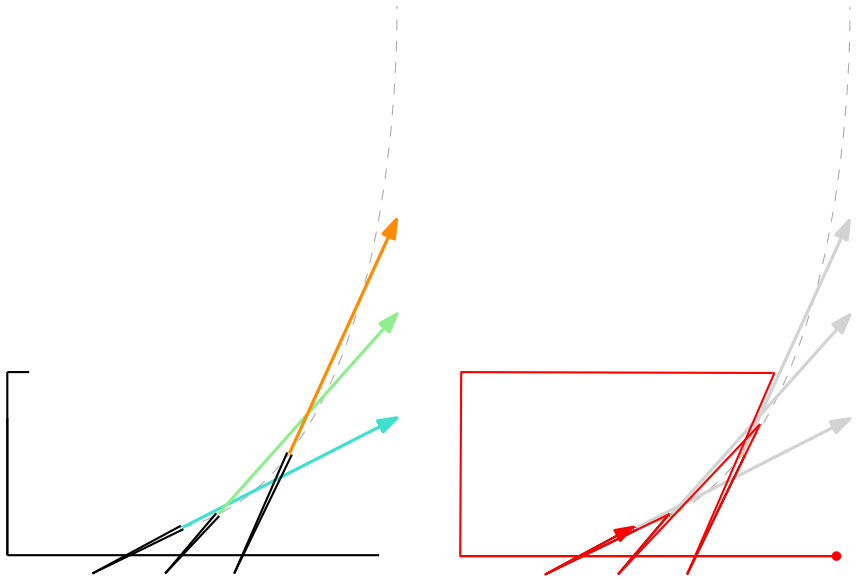


Figure 3.20: (a) Sketch of a reduction of a circle graph with three chords. Segments shown in black. (b) Polyline of $2n + 3$ links covering the constructed segments, corresponding to a Hamiltonian path traversing the rays, starting with the ray with the largest slope and ending with the ray with the smallest slope.

horizontal, covering s_t. Since the supporting line of s_t intersects all of the rays, the polyline can bend to any needle segment for its next link.

Since we have covered our additional segments s_h, s_v, and s_t, the rest of the $2n$ links must cover one needle segment each. Observe that the needle segments all extend downward to below the lowest intersection point between supporting lines. This means that when a link covers a needle segment when travelling downward, the next link must then travel upward on the other half of the needle, as all intersection points with supporting lines of other segments lie in the segment's interior. When the next link then covers a needle segment when travelling upward, the only places the polyline can viably bend next are near places where the ray associated with the previous needle intersects another ray. So we can cover the $2n$ needle segments using a polyline of $2n$ links if and only if there is a Hamiltonian path in the ray intersection graph and thus a Hamiltonian path in the original circle graph. □

Since the transformation is polynomial, we know the problem is NP-hard. We can also see that the problem is in NP, since for any instance we can expect that if a polyline of k links exists covering a set L, one must also exist where each vertex has coordinates of polynomial complexity, since the vertices could all lie on the intersection points of the supporting lines of the segments, or otherwise on points with rational coordinates on those supporting lines. This polyline could serve as a certificate for the verification algorithm. This gives the following theorem:

Theorem 3.10.6. *The connected segment polyline cover problem is NP-complete.*

▸ 3.10.3 Reducing to global curve simplification

Finally, we reduce the connected segment polyline cover problem to non-restricted GCS using the Hausdorff distance from P to P'.

As a problem instance, we are given a set L of n non-collinear line segments in the plane whose union is connected. We construct an input polyline of polynomial size that completely covers the set of segments and no other points. We could do this, for example, by treating the segment endpoints and intersection points as vertices of a graph connected by edges, and have our polyline be the path of a breadth-first search through the graph visiting all vertices. We set δ to 0. Now we know that, since a simplification must cover the union of L, any simplification of our input polyline that has n links must cover each segment in L completely with one link. This means such a simplification would be a solution to our instance of connected segment polyline cover. Since the reduction is polynomial in size, we know that this variant of the GCS problem is NP-hard, and using a similar argument to the one for the connected segment polyline cover problem it is easy to see that it is in NP as well.

Theorem 3.10.7. *Non-restricted global curve simplification under the Hausdorff distance directed from the original curve to the simplification, restricted to instances where $\delta = 0$, is NP-complete.*

▸ 3.10.4 Non-zero δ

We can also extend this reduction to non-zero δ by picking $\delta > 0$ but still small enough such that it would not change the combinatorial structure of the space the polyline can lie in, so each link of the polyline must still correspond to exactly one segment in L. In the remainder of this section, we will treat δ as an unknown and aim to determine a value where we can guarantee any δ smaller than that value will not change the combinatorial structure.

First, consider that we have constructed $2n + 3$ segments. For $\delta = 0$, the space the polyline can lie in is exactly these segments, but for non-zero δ, this space is the result of replacing each segment by the Minkowski sum of the original segment and a disk of radius δ. This means that the polyline does not have to exactly cover the original segment, meaning additional angles of approaching a segment open up. If we center two circles with radius δ on the endpoints of a segment, the two inner tangents of these circles will form the bounding lines of a cone that covers all possible polyline links that are able to "cover" a segment. We will call the part of the cone that is within δ of the segment the *tip* of the cone, and the rest of the cone the *tail* of the cone. For $\delta = 0$, the supporting line for the segment forms a degenerate cone of width 0. To preserve the combinatorial structure, fattening the cones cannot introduce intersections between cone tails, as these correspond to two segments' supporting lines intersecting in the exterior of the segments. We do not have to consider the cones associated with the leading segments, since they already intersect all of the needles' cones in the tip. So we will focus on the cones associated with needle segments.

Instead of taking the actual cones and the half-lines that bound them, we will consider a larger cone that is easier to calculate with. As bounding lines, we will use endpoints created by going 2δ to the left and right of the original endpoints of the needle segments. For a ray starting at point $(a, a!)$, the original endpoints for one of its needle segments are (x_ℓ^a, y_ℓ) and a point very close to $(a, a!)$, where x_ℓ^a is the x-coordinate of the supporting line of the ray at $y = y_\ell$. This gives the segment a slope of $\frac{a!-y_\ell}{a-x_\ell^a}$. The slope of the left bounding line of the cone is then $\frac{a!-y_\ell}{a-x_\ell^a-4\delta}$. So, departing from the intersection point of the bounding line and the needle segment, for every $(a! - y_\ell)$ we move upwards, the bounding line lies 4δ further to the left as compared to the needle segment and/or the ray associated with the needle.

To see how small δ needs to be, we consider any two rays where one ray connects the points $(a, a!), (b, b!)$ and one ray connects the points $(c, c!), (d, d!)$. We will refer to the cones induced by their needle segments as the (a, b)-cone and (c, d)-cone respectively. We assume $a < c$. Now there are three cases again analogous to those in the proof for Lemma 3.10.2, see Figure 3.21:

- The case where $c < b < d$. Here, the tails of the cones already intersect, so no δ is going to introduce a new intersection.

- The case where $b < c$. Here, there is already an intersection between the tail of the (a, b)-cone and the tip of the (c, d)-cone. It is important that after fattening the cones there is no intersection that lies in both tails. At $y = c!$ we know that the right bounding line of the (c, d)-cone has shifted 2δ to the

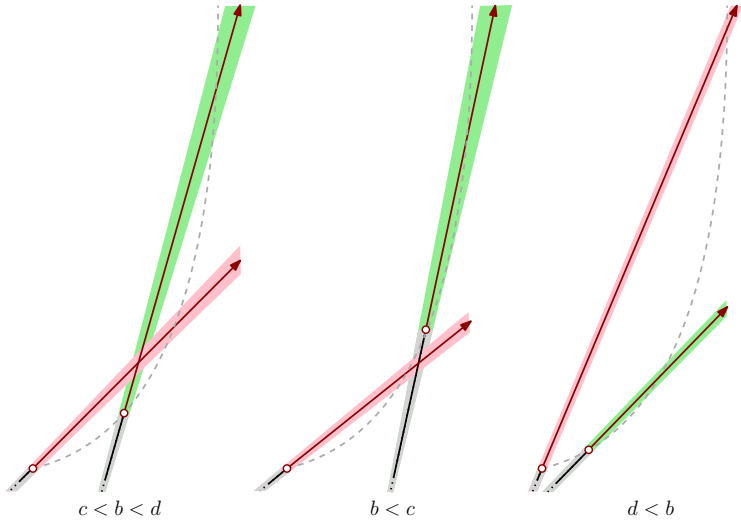


Figure 3.21: The three cases for two needle segments' cones. The segments are shown in black. The tips of the cones are shown in gray. The tail of the (a, b)-cone is shown in pink and the tail of the (c, d)-cone in green. For the second and third case, the cone can be fattened (i.e. δ increased) as long as this does not cause the tails to intersect.

right of where it would be for $\delta = 0$. Likewise, the left bounding line of the (a, b)-cone has shifted $4\delta \cdot \frac{c!-a!}{a!-y_\ell}$ leftwards. If the total shift is less than the original horizontal distance between the rays at this y-coordinate, we know the intersection between the cones still lies in the tip of the (c, d)-cone. The horizontal distance between the unshifted lines at this value for y, $dist_h$ can be written as

$$dist_h = a + \frac{c! - a!}{b! - a!} \cdot (b - a) - c$$

For a fixed b, we get a smaller value the higher a is, as this increases the slope of the (a, b)-cone. The value is also smaller the smaller c is, as this decreases the vertical distance and therefore the horizontal distance to the point $(c, c!)$ as well. Substituting $a = b - 1, c = b + 1$ into the equation gives $dist_h = \frac{b^2}{b-1} - 1$. This is smallest when b is minimal, so the minimum distance of 3 is achieved when $b = 2$. This means that a bound on δ is

$$3 > 4\delta \cdot \frac{c! - a!}{a! - y_\ell}$$

We can replace the right hand side of this inequality by an upper bound

$$3 > 4\delta \cdot n!$$

This gives us an upper bound on the value for δ of $\delta < \frac{3}{4n!}$.

- The case where $d < b$. Here there is an intersection of the cones in the tips. We need to make sure not to introduce new intersections of the tails. For this, the left bounding line of the (c, d)-cone needs to intersect the right bounding line of the (a, b)-cone in the tip. We know that at $y = a!$, which is near where the tip of the (a, b)-cone ends, the right bounding line has shifted exactly 2δ to the right compared to when $\delta = 0$. At $y = c!$, the left bounding line of the (c, d)-cone will have shifted 2δ to the left, so at $y = a!$ we know that the total shift is less than 4δ. The horizontal distance $dist_h$ between the supporting lines of the two rays at $y = a!$ is equal to

$$dist_h = c - \frac{c! - a!}{d! - c!} \cdot (d - c) - a$$

For a fixed c, the smallest distance is obtained when $a = c - 1$ (as increasing the value of a decreases the vertical distance between c and a meaning they are also closer horizontally at $y = a!$), and $d = c + 1$ (as decreasing d decreases the slope of the (c, d)-cone, and so the supporting line will be further left at $y = a!$). Plugging these values into the equation for $dist_h$ lets us rewrite it to

$$dist_h = 1 - \frac{c - 1}{c^2}$$

Since $c \geq 2$ in this case, we know the smallest possible distance is $\frac{3}{4}$. Therefore, we know this case will pose no problems if $4\delta < \frac{3}{4} \longrightarrow \delta < \frac{3}{16}$, as the shift is smaller than the original distance so the intersection of the cones still lies in the tips.

Over all three cases, the bound $\delta < \frac{3}{4n!}$ is smallest and so it is the one that should be used. This upper bound is not necessarily tight, but we do now know that we can find non-zero values for δ that can be represented in a polynomial number of bits that do not change the combinatorial structure of the input for the simplification problem. So a simplified polyline of $2n + 3$ links with $0 < \delta < \frac{3}{4n!}$ will only exist if and only if it also exists for $\delta = 0$. Since we can have a small enough δ of polynomial bit complexity, this means this variant of GCS is hard in general, as for larger values of δ the construction could be scaled up.

If the general problem is in NP is hard to say, since our approach for showing this for the $\delta = 0$ case does not extend, and it might be possible that inputs exist where the only possible simplified polylines of k links have vertex coordinates of exponential bit complexity. We leave this as an open problem.

Theorem 3.10.8. *Non-restricted global curve simplification under the Hausdorff distance directed from the original curve to the simplification, is NP-hard.*

▸ 3.11 Conclusion

In this chapter, we systematically studied the global curve simplification problem under different (global) distance measures and constraints in which vertices can be placed. We improved some of the existing results in the vertex-restricted case and we obtained the first NP-hardness results for the curve-restricted version. In the future, providing approximation algorithms, particularly for the curve-restricted case, can be of interest.

Algorithms for the vertex- and curve-restricted variants of the problem can straightforwardly be applied to trajectories. The vertices of the simplification either correspond to probes of the original trajectory, or in the curve-restricted case they correspond to constructed probes that can be obtained by linear interpolation. Global methods are less applicable to the trajectory case than local methods, however, since the global methods take only the final shape into account and can simplify away sections where the trajectory doubles back on itself. When timestamps are then assigned to the vertices of the simplification the speeds indicated by the simplified trajectory may be very different to those actually occurring for the entity. For the non-restricted case there is no straightforward way of assigning timestamps to the vertices of the simplification, so these methods are probably better left to the non-trajectory case where only the shape aspects of the curve matter.

Chapter 4

Representative Trajectories

4.1 Introduction

One *tool* that can be used to bring order to a large trajectory database is trajectory clustering: Trajectories that are spatially and/or temporally close are grouped together. This can help focus application task algorithms that are applied later, as they do not have to consider the whole set of trajectories when searching for patterns, but only the trajectories within each cluster where there is a guarantee that they lie close together. Looking at a visualization of the biggest clusters in a data set is also a good way of getting an initial idea of which movements are most common in the data set. Another tool that can be used to gain insight into the characteristics of clusters, is to assign each cluster a trajectory to be its representative. These representative trajectories should give a good indication of the contents of the clusters. Representative trajectories can either be picked from the trajectories in the cluster, or specifically constructed to be a good representative for the cluster, especially if there are no good candidates to choose in the cluster. See Figure 4.1.

One method for generating a representative trajectory is the *central trajectories algorithm* (CTA), introduced by Van Kreveld et al. [90]. To the best of our knowledge, this method has only been studied theoretically until now and has never been implemented. In this chapter, we present the first implementation of this algorithm and experimentally study it to see how real world results compare to theoretical ones, particularly when it comes to output complexity. The CTA requires each trajectory in the cluster to have the same complexity. The CTA has a theoretical worst-case output complexity of $O(\tau n^{5/2})$, where τ is the number of links per trajectory and n

is the number of trajectories in a cluster. So it is theoretically possible that the generated representative trajectory has a complexity that is many times greater than the complexity of the input trajctories, which is unwanted. However, the upper bound on output complexity is reached when the generated representative trajectory follows a pathological zigzagging pattern, which seems unlikely to occur in practice, see Figure 4.2. With this research, we aim to find out what kind of output complexity we can actually expect when applying the CTA to real-world trajectory clusters. We also look at the influence of adding trajectory simplification when computing representative trajectories, as trajectory simplification is one possible way of reducing the complexity of the representative trajectory. We examine the effects on the complexity of the representative trajectory of simplifying either the input trajectories and/or the representative trajectory itself.

This chapter is organized as follows: In Section 4.1.1 we discuss the relevant literature. In Section 4.2 we explain the algorithms we use in more detail. In Section 4.3 we discuss the setup for our experiments. In Section 4.4 we present our results, and we conclude in Section 4.5.

▸ 4.1.1 Related work

Trajectory clustering Data clustering is a technique commonly used in data mining, where data is split up among clusters such that each data item has more in common with data of its cluster than with data in other clusters. For trajectory data, clustering can be of vital importance, as the movement patterns to be found in trajectory data mining often occur on a local scale, within a limited area or timespan. So it does not make sense to examine an entire database at once, where trajectories may lie far apart in space and time. Traditional data clustering techniques such as k-means clustering and DBSCAN [17, 18] have been adapted for use on trajectories. But there have also been techniques introduced specifically for clustering trajectories.

Lee et al. [93] partition trajectories into line segments, and then apply clustering on these line segments. Gariel et al. [61] cluster aviation trajectories by simplifying them and then applying a Longest Common Subsequence (LCSS) algorithm. Gaffney et al. [59] use a probabilistic clustering method known as a finite mixture model, where each cluster gets a probability density function (PDF) that is learned with an expectation-maximization algorithm. Clusters can then be assigned based on which PDF gives the highest probability for a trajectory. Fu et al. [58] introduce spectral clustering for car trajectories extracted from video footage: They construct a similarity matrix between the trajectories and then cluster by finding a partition

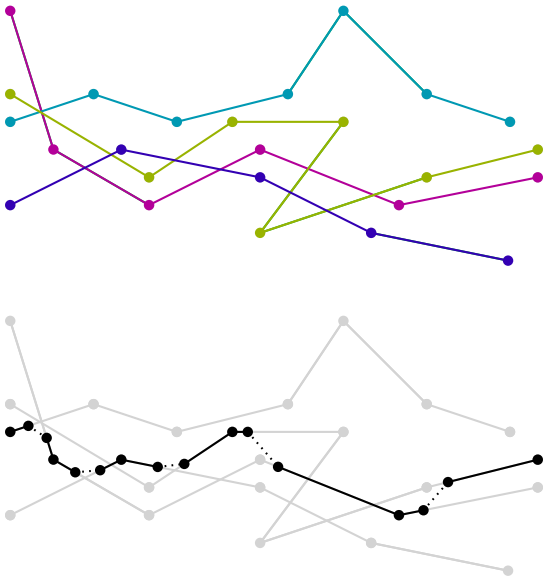


Figure 4.1: Top: A trajectory cluster where no trajectory is a good representative for the cluster. Bottom: A trajectoid constructed to be representative of the cluster. The dashed sections represent discontinuities where the trajectoid switches between trajectories. This can only occur when the points on both trajectories corresponding to the time of the switch are closer than the distance threshold ε.

of the graph induced by the matrix. Zhang et al. [138] compare different clustering algorithms and also introduce a metric for measuring the similarity in a cluster and disparity between clusters.

The clustering algorithm used for extracting clusters from the used data sets in this chapter is the *Subtrajectory Clustering* algorithm introduced by Buchin et al. [32]. For details on this algorithm and how we have adapted it for use in this chapter, see Appendix B.

Cluster summarization Once clusters have been formed, it is beneficial to summarize the clusters so they can be visualized and compared more easily. In this chapter we study the algorithm of Van Kreveld et al. [90], but other methods also exist. Some methods are based on computing which trajectory in a cluster best represents the cluster [114], but this leaves open the possibility that none of the

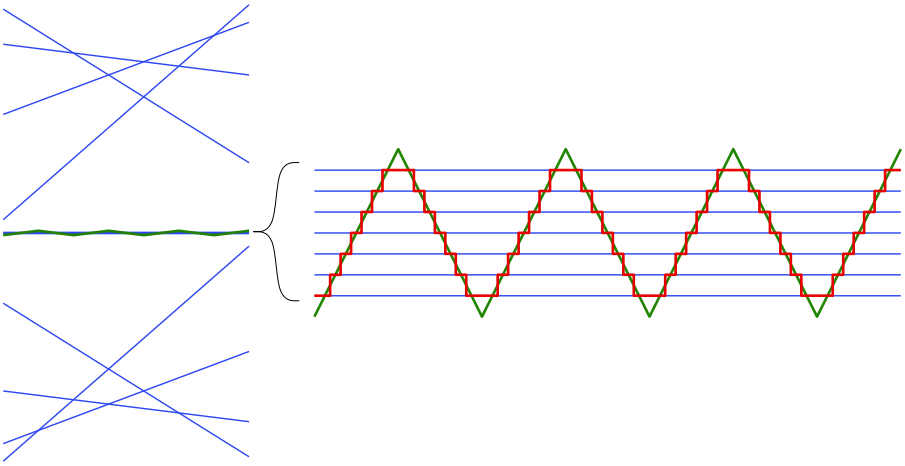


Figure 4.2: Pathologic case for 1-dimensional trajectories. Time is visualized as the horizontal axis. Within a time interval in between probe sampling times, n trajectories (blue) can be arranged to make the ideal representative trajectory (green) zig-zag, giving the central trajectory (red) a complexity of $O(n^2)$. This construction can be copied in between each pair of probes giving the central trajectory a total complexity $O(\tau n^2)$. For 2-dimensional trajectories the complexity in between a pair of probes can become $O(n^{5/2})$. Figure adapted from [90].

trajectories represent the total cluster well enough. So most methods create wholly new trajectories that are not part of the cluster, but do represent it.

Buchin et al. [33], only consider the spatial aspects of the trajectories and discard the time component. They then use the median trajectory as a representative. This is a *trajectoid* (see Section 4.2 for the definition) that is optimized to be the median of the cluster at all times, i.e. from a point on the median trajectory, for the point to reach the unbounded face it must cross at least half of the trajectories. They also introduce a version where the median trajectory must preserve homotopy with the cluster respective to a set of obstacles. Lee et al. [93] compute a representative trajectory based on the averaged position of the cluster. Johard and Ruffaldi [80] compute a mean trajectory for analyzing human body movements using a generalization of Dynamic Time Warping barycenter averaging.

Trajectory simplification In this chapter, for our experiments where we test the effects of trajectory simplification, we use the widely-used Douglas-Peucker [49]

algorithm for simplification. It uses a divide-and-conquer algorithm for simplifying polylines. The algorithm ensures the Hausdorff distance between original and simplification is at most a parameter δ_{simp} but it offers no guarantee of optimal data reduction. For more related work on trajectory simplification we refer to Chapter 3.

4.2 Preliminaries and notation

4.2.1 Trajectory clusters

A trajectory *cluster* C is a set of (sub)trajectories close in space and time. For the purposes of this chapter, we define it as a set of at least m subtrajectories, such that each subtrajectory has distance of at most 2δ to any other subtrajectory in the cluster, where m and δ can be chosen based on the use case. How large the distance between trajectories is will depend on the distance measure that is used. Many different measures exist, and we define several in Section 1.2. For our clustering in this chapter, we use the algorithm by Buchin et al. [32] which can use either the Fréchet distance or discrete Fréchet distance. (See Section 1.2.2.) We use the version based on discrete Fréchet distance so we can make use of the implementation present in the library MoveTK [2]. We assume that the trajectories in a cluster are all sampled at the same times and thus also have an equal number of edges. We notate this number of edges as τ.

4.2.2 Trajectoids

A *trajectoid* is a function on a cluster of trajectories that maps time to these trajectories. It can be thought of as a pseudo-trajectory consisting of a concatenation of pieces of the input trajectories. A trajectoid is said to be ε-connected if, at each time where the trajectoid switches which trajectory piece is followed, the Euclidean distance between the old and new pieces at that time is at most ε. The places where the trajectoid switches trajectory pieces are called *discontinuities*. See Figure 4.1. Note that trajectoids take the time component of the input trajectories into account. So even if two trajectories lie close to each other, the trajectoid can only switch which trajectory it follows if the entities were close enough to each other at some specific time.

4.2.3 Central trajectories

Let $D(\mathcal{T}, t)$ be the radius of the smallest disk centered on trajectoid $\mathcal{T}$ at time t that encloses all entities in the cluster. The *central trajectory* is then the ε-connected trajectoid that minimizes the function

$$\mathcal{D}(\mathcal{T}) = \int_0^{t_\tau} D(\mathcal{T}, t)\mathrm{d}t \tag{4.1}$$

Informally, it is the trajectoid that is as close to the center of the cluster as possible while respecting ε-connectedness. The *central trajectories algorithm* (CTA) as posed by Van Kreveld et al. [90] takes a parameter ε. It works by constructing a Reeb graph. A Reeb graph is a graph that captures the sequential merging and splitting of sets. Each edge (called a Reeb edge) corresponds to a set, and a vertex corresponds to a change. If a vertex has two incoming edges and one outgoing edge, it represents the merging of two sets. Inversely, if one edge comes in and two go out, the set is split in two. The Reeb graph constructed by the CTA is one where each edge corresponds to a maximal ε-connected subset of the cluster, and each vertex corresponds to an event where these sets change. So these are times where two entities are at a distance of exactly ε and the set containing one of the entities splits from or merges with the set containing the other entity. There can be a quadratic number of these vertices. The edges are assigned a weight based on the centrality of the trajectories associated with the edge. The central trajectory is then found by taking a minimum weight path through the Reeb graph. See Figure 4.3 for a 1-dimensional example of a Reeb graph constructed like this, and Figure 4.4 for such a Reeb graph annotated with the centrality functions that will be the basis for its edge weights. For the full details of the central trajectories algorithm, we refer to the paper by Van Kreveld et al. [90].

4.3 Experimental setup

In order to verify the output complexity and effects of simplification when using real data, we design two experiments. In the first experiment we vary the ε parameter for the CTA and run it on several real data sets and study how the output complexity changes as ε increases. In the second experiment we fix ε and run the CTA four times on the data sets, testing different combinations of applying a simplification algorithm before or after running the CTA. We repeat this for three different levels of simplification. To perform the experiments, we need:

- An implementation of the central trajectories algorithm.

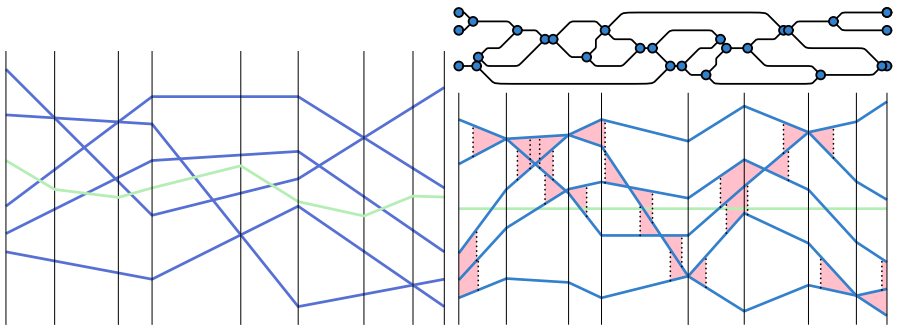


Figure 4.3: (left) A set of five 1-dimensional trajectories of four vertices each (blue). The x-axis corresponds to time. The *ideal* trajectory which minimizes the sum of distances is shown in green; breakpoints of the ideal trajectory are indicated by vertical black lines. (right) An equivalent transformed version of the trajectories in which the ideal trajectory is a straight line. Times at which two or more trajectories are within distance ε are indicted in pink. The resulting topological structure is captured by the Reeb graph, drawn at the top. Figure taken from [90].

- An implementation of a simplification algorithm.
- Suitable real-world data sets.
- Preprocessing and clustering of the data sets.

4.3.1 Implementation choices

We have implemented the CTA using C++ and making use of the libraries CGAL [3], Boost [4], and MoveTK [2]. As a simplification algorithm we use the implementation of Douglas-Peucker from MoveTK and for the clustering algorithm we have adapted the implementation of Subtrajectory Clustering that is included in MoveTK.

4.3.2 Data sets

The input for the central trajectories algorithm should be a clustered set of trajectories. We obtain such clusters from several data sets coming from different domains. We use the HR, MC, TD, GL, and PF data sets. See Appendix A for details on the data sets. For the CTA, input data needs to be in a clustered form, so we have done a number of preprocessing steps to extract trajectory clusters. See Appendix B for details.

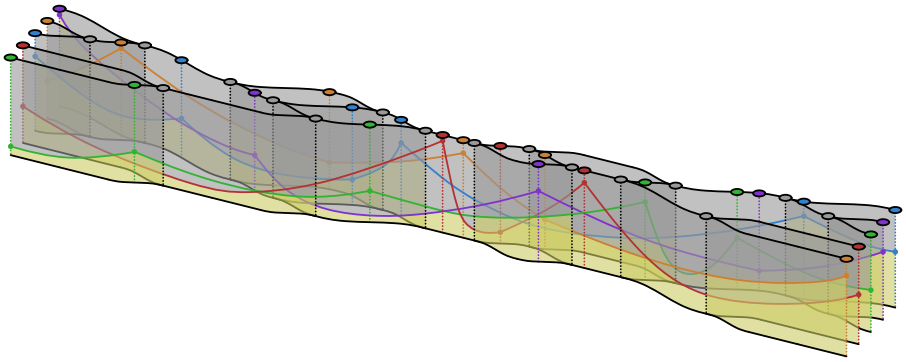


Figure 4.4: A Reeb graph for five entities annotated with the functions for each entity giving the distance to the centerpoint of the smallest enclosing disk. The weight of each Reeb graph edge is based on the lower envelope of the functions on that edge. Figure taken from [90].

4.3.3 Final data sets

After the preprocessing we obtained five data sets, each containing 94–160 clusters of between 3 and 31 trajectories with 9–1226 probes. This will be the input for the CTA. However, a sizable amount of these probes are colinear, introduced by the elementary resample done during preprocessing (see Appendix B), as well as incidentally colinear probes. For comparing the input and output complexity these colinear probes are considered to be noise, so when reporting the results we give the input complexity for the trajectories after they have been simplified using Douglas-Peucker with a very small distance threshold δ_{simp} of 1 cm to remove the colinear probes. Using this approach, the complexity of the trajectories in the input comes to lie between 2–463 probes, so this removal of probes does mean the input complexity can become lower than $|T|_{min}$, i.e. the minimum complexity parameter used during preprocessing (see Appendix B).

Examples of clusters found in the data sets can be seen in Figure 4.5. Although we have taken steps to exclude stay points from our clusters, some are still present in our preprocessed data. This is particularly true for the TD and PF data sets. Whereas most of the data sets contain many clusters where the trajectories follow along the same road (or shipping channel), TD mostly lacks these clusters. This is likely because of TD's relatively low sample rate. Most of its clusters are simply subtrajectories lying close to some point, without necessarily going the same way.

Important metrics about the clusters are given in Table B.2.

Figure 4.5: Example clusters of the data sets, from left to right: Top row: HR, TD. Bottom row: GL, PF, MC. The approximate size of the bounding boxes for these clusters are, respectively: $4.2km \times 2.9km$, $1.6km \times 1.5km$, $1.3km \times 2.1km$, $6.3km \times 14.2km$, $29km \times 29km$.

▸ 4.3.4 Additional details on the implementation of central trajectories

The edge weights for the Reeb graph used by the central trajectories algorithm are based on solving an integral. For ease of implementation we approximated the intergrals using a Riemann sum. The number of rectangles used for the approximation is equal to $\frac{\ell(I_e)B}{t_\tau}$, where $\ell(I_e)$ is the length of the time interval during which edge e is active, t_τ is the total time the cluster is active, and B is a parameter setting the maximum number of rectangles. The B sample times are also the times where we evaluate if the central trajectory should switch which trajectory in the edge is used. This means that the central trajectory can switch mid-edge at most B times. Of course the trajectory can also switch at the end of edges. In our experiments, $B = 1000$. Since we will show in our experiments that the number of discontinuities is comparatively small, we are confident that this is a large enough value for

this parameter.

4.3.5 Experiment 1: output complexity of real data

We applied the central trajectories algorithm to the data sets while varying the values for ε from 0 to 100 meters in increments of 5. We measured the number of output vertices and the number of discontinuities (i.e. the number of times the central trajectory changes which trajectory it follows). For the data sets where the average value of these measurements had not yet converged at an ε of 100 meters (TD and MC) we repeated the experiment but now varying ε from 0 to 1000 meters in increments of 50.

4.3.6 Experiment 2: effect of simplification

We also experimented to investigate the effect of trajectory simplification on the results. To do this, we applied the CTA with a fixed ε of 100 meters. For each data set, we apply the CTA for four different combinations of applying the simplification algorithm:

- No simplification.
- Simplifying the input.
- Simplifying the output.
- Simplifying both input and output.

We run this experiment for three values of the simplification parameter δ_{simp}, namely 1, 10, and 100 meters. We measure the complexity of the central trajectory we get after all of the combination's simplification steps are applied.

4.4 Results and discussion

The output data for our experiments that we have plotted below can be found in [5].

4.4.1 Results of experiment 1

Visualization of results First we will consider the results of the first experiment, as seen in Figure 4.6. We have plotted one curve for each cluster, showing how the

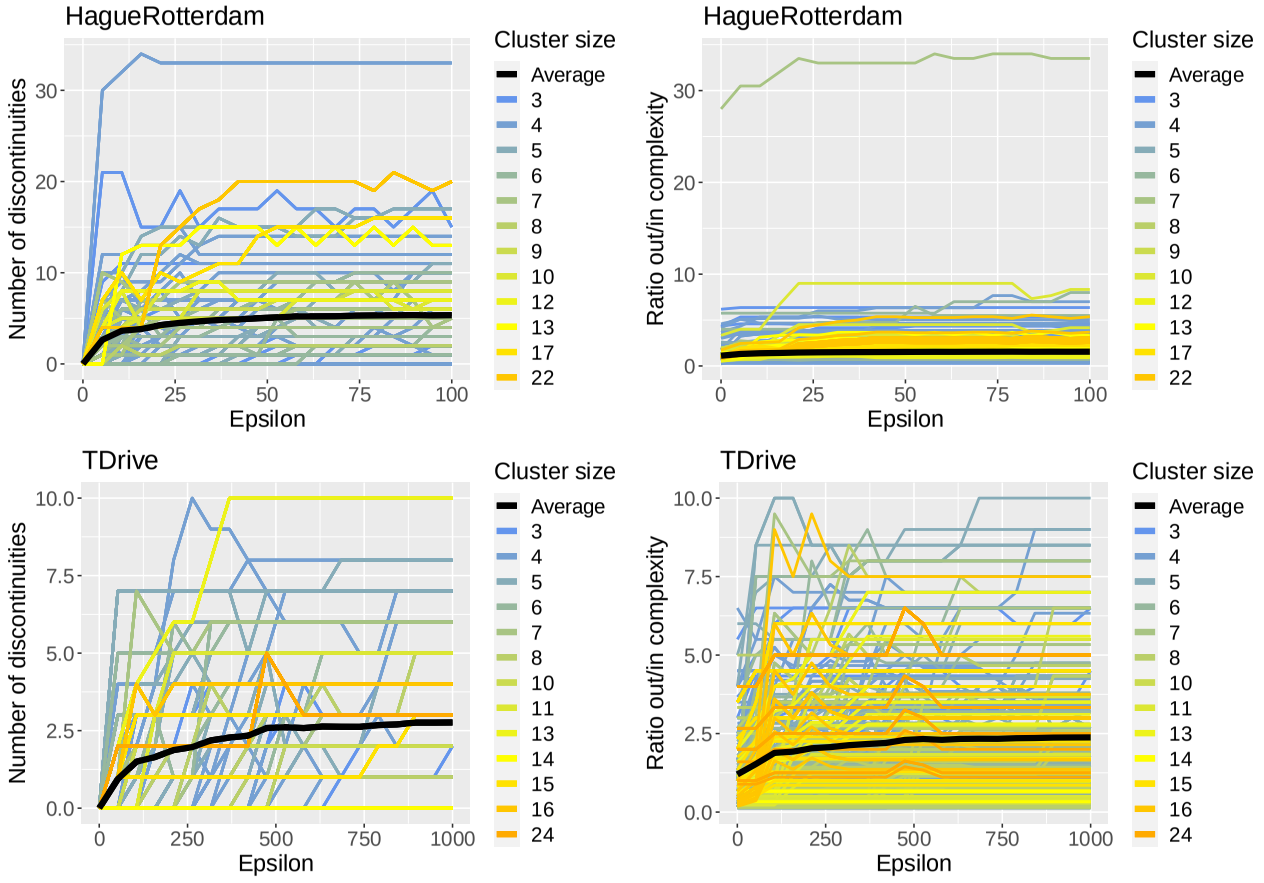


Figure 4.6: Results of Experiment 1. Each curve represents one of the clusters of the data set (continued on the next page).

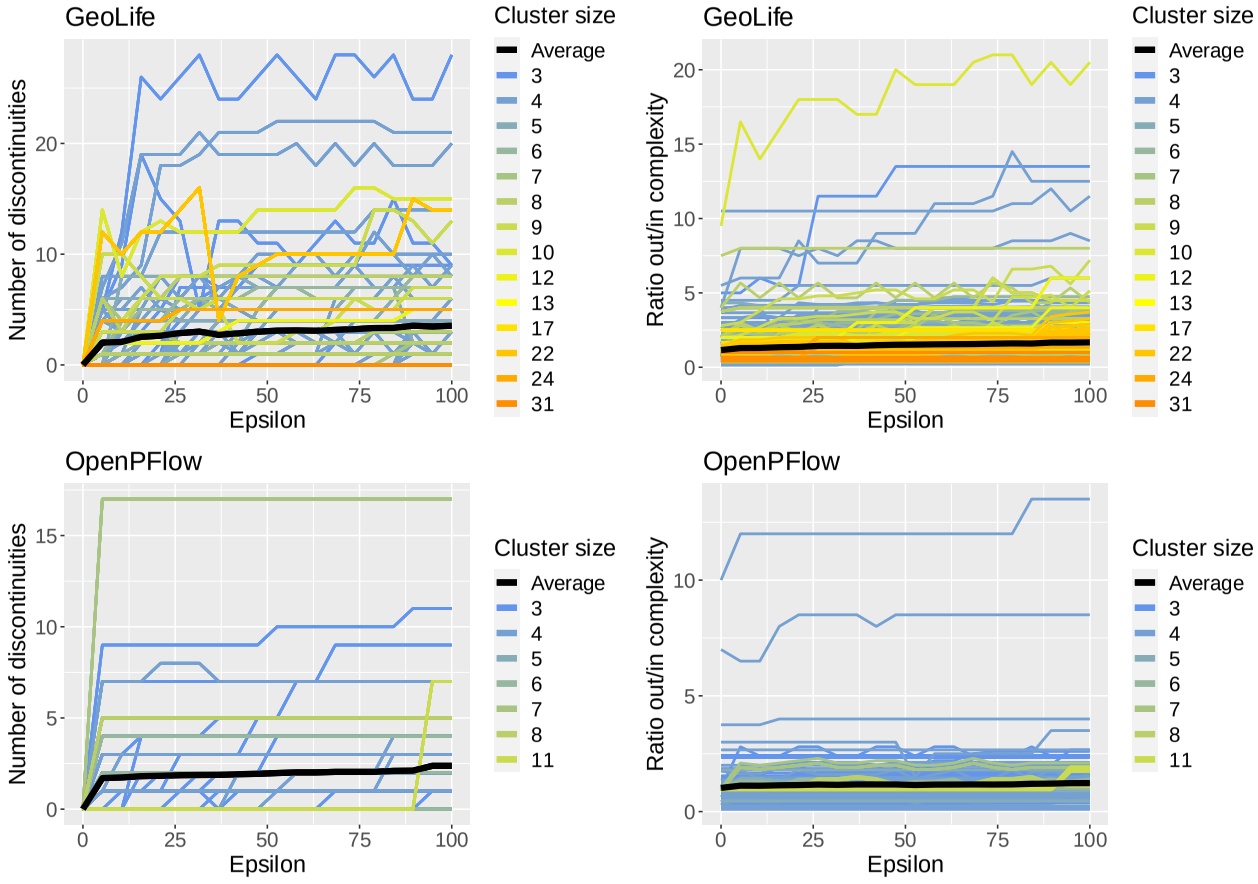


Figure 4.6: Results of Experiment 1. Each curve represents one of the clusters of the data set (continued on the next page).

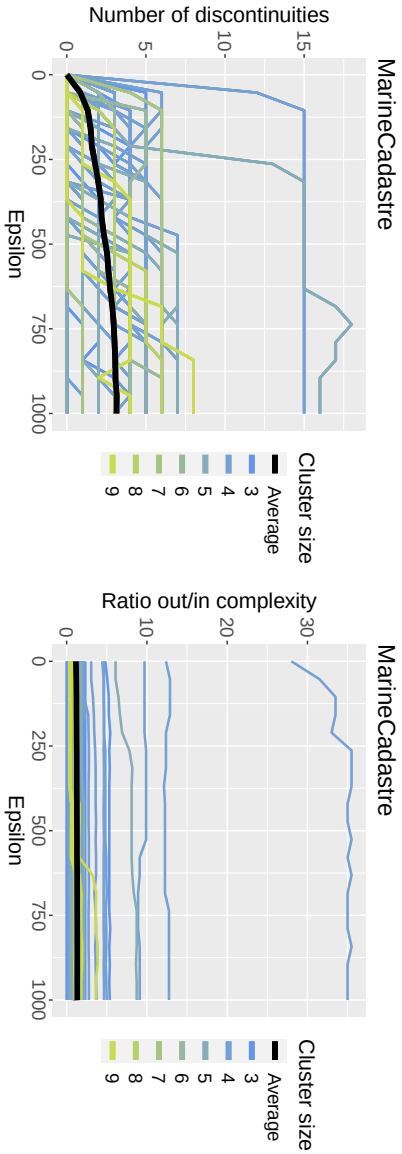


Figure 4.6: Results of Experiment 1. Each curve represents one of the clusters of the data set.

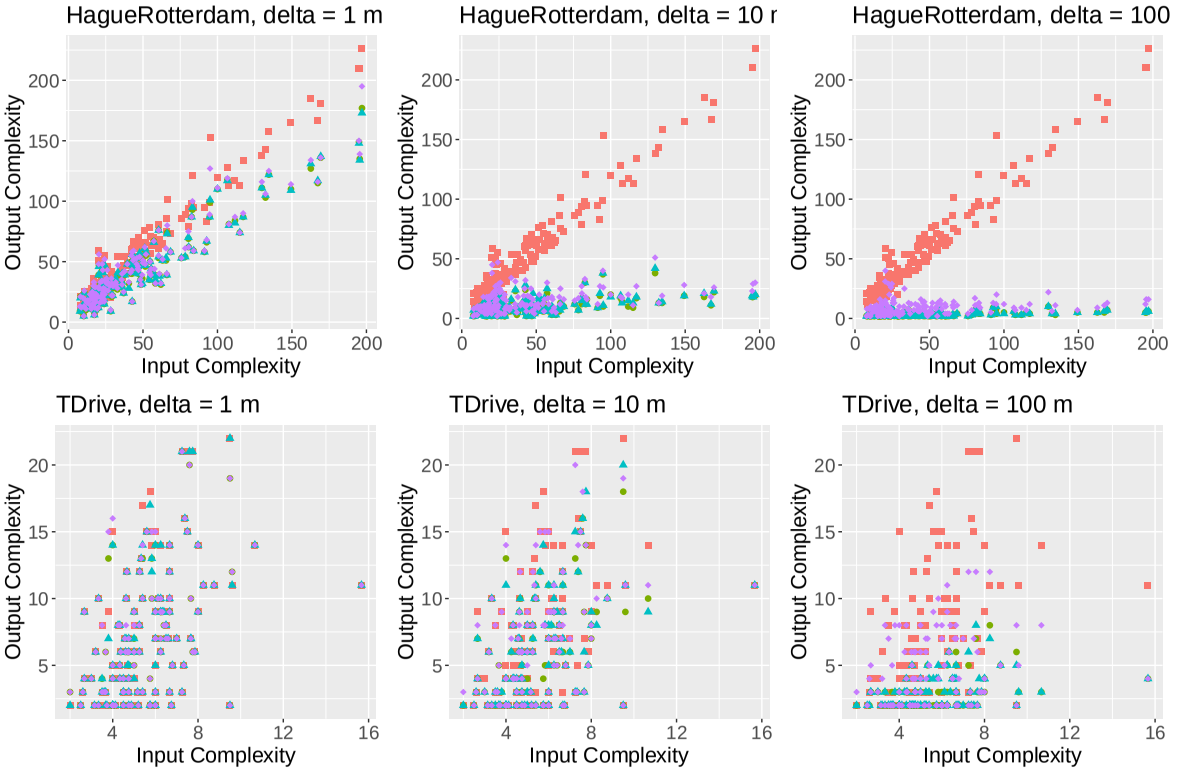


Figure 4.7: Results of Experiment 2. Red squares represent no simplification applied. Pink diamonds were simplified before the CTA was applied. Blue triangles were simplified afterwards and green circles were simplified both before and after the CTA (continued on the next page).

Figure 4.7: Results of Experiment 2. Red squares represent no simplification applied. Pink diamonds were simplified before the CTA was applied. Blue triangles were simplified afterwards and green circles were simplified both before and after the CTA (continued on the next page).

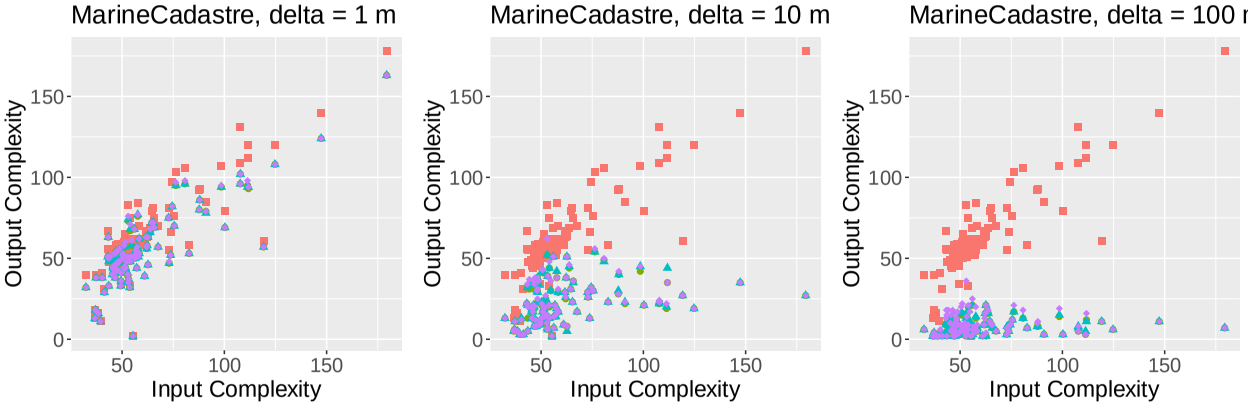


Figure 4.7: Results of Experiment 2. Red squares represent no simplification applied. Pink diamonds were simplified before the CTA was applied. Blue triangles were simplified afterwards and green circles were simplified both before and after the CTA.

number of discontinuities and the ratio of output to input complexity changes as we increase ε. Each curve is colored according to the number of trajectories in the cluster. Since the input complexity for the trajectories in the cluster may vary a bit due to the removing of colinear probes the ratio of output to input complexity is averaged over all trajectories in the cluster. We also show the mean average over all clusters for each value of ε as a black curve.

Interpretation of results. We can see that up to a certain point, as ε increases so does the number of times the central trajectory switches which input trajectory it follows and so does the ratio between output and input complexity. However, once ε is big enough the curves level out. Since the CTA optimizes for centrality and not for reducing the number of discontinuities, a larger epsilon can also result in a reduction of discontinuities. Because of this, some clusters' curves tend to oscillate around a certain value as ε increases.

By looking at the cluster size of the trajectories, we can see that the ratio of output to input complexity does not seem to be closely related to the cluster size, depending considerably more on the individual trajectories. The ratio generally lies close to 1, even for most of the larger clusters, not becoming larger than 4. The theoretical worst case output complexity is $O(\tau n^{5/2})$, which would mean a possible ratio of $n^{5/2}$, so for large clusters we could expect ratios orders of magnitude greater than what we see in practice. Thus, we have shown that this theoretical worst case is not representative of the output complexity in practice.

▸ 4.4.2 Results of experiment 2

Visualization of results. Now we will look at the results of the second experiment, shown in Figure 4.7. The scatterplots show four points for each cluster in different shapes and colors: A red square indicating the input and output complexity when no simplification is applied, a pink diamond for when trajectory simplification is applied to the input trajectories but not to the output, a blue triangle for when the output is simplified but not the input, and a green circle for when both input and output are simplified.

Since the input complexity for the trajectories in a cluster may vary a bit due to the removing of colinear probes the input complexity is averaged over all trajectories in the cluster.

Interpretation of results We can see that the impact of simplification for a given value of δ_{simp} depends on the data set. For a δ_{simp} of 100 meters we do see a signifi-

cant impact for all data sets. We can see that the red points, corresponding to when no simplification is applied, generally stay close to the line $x = y$, which confirms our findings of the previous experiment that the ratio between output and input complexity tends to stay close to 1 even for large clusters. Looking at the difference between the purple, blue and green points we can see that simplifying the output rather than the input gives a slightly lower output complexity. The extent to which they differ depends on the data set and the choice for δ_{simp}. The difference between only simplifying the output, and simplifying both the output and input is very small, as the green and blue points lie very closely together. So for reducing the total output complexity simplifying the output suffices. Simplifying the input complexity is more useful for reducing the memory required to store the input clusters. Of course, simplifying either the input or the output changes the properties of the trajectories, so the central trajectory may not be optimal anymore for the original cluster.

Artifacts Upon close inspection, we can see that the PF data set has one interesting outlier when the input is simplified with $\delta_{simp} = 100$. There is one cluster with output complexity of 141, placing it far above the other points in the graph. Investigating this outlier reveals that after simplification, the three trajectories in the outlying cluster have one segment each that lie nearly equally close to their center. The output central trajectory rapidly jumps between the segments creating very high complexity for this one set of segments. See Figure 4.8.

This seems to be a rare occurrence. In particular, this is not an example of the pathologic zig-zagging described by Van Kreveld et al.; in fact, the output complexity is actually larger than the worst-case bound from [90]. We believe this is an artifact caused by rounding errors in our Riemann approximation (refer to Section 4.3.4). We verified this by varying the value for the number of rectangles B, and indeed the output complexity of this instance changes accordingly. We expect that using exact computation for the Reeb graph edge, rather than the Riemann approximation, would resolve this issue.

▸ 4.5 Conclusion

In conclusion, we have conducted the first experimental study into the central trajectories algorithm. We show that, on most clusters in practice, the output complexity does not reach the theoretical worst case of $O(\tau n^{5/2})$, (where n is the number of trajectories in the cluster and τ is the complexity of each input trajectory). Instead, the output complexity is closer to the number of vertices of an input trajectory in most cases, meaning that the degenerate patterns responsible for this bound do not

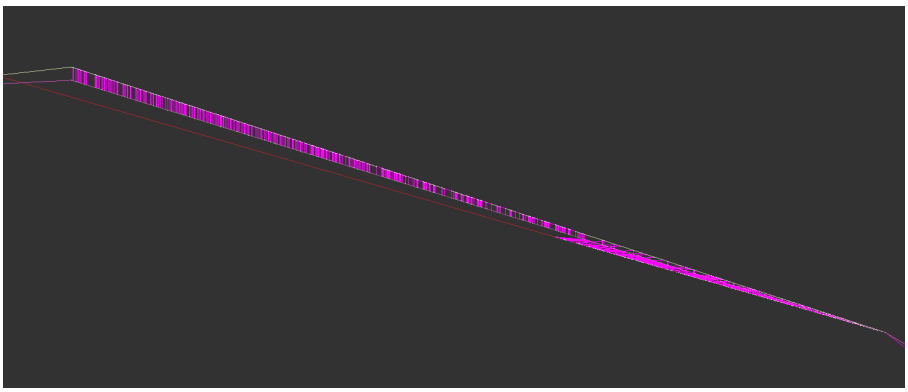


Figure 4.8: The outlier found in OpenPFlow. Central trajectory shown in pink. The amount of discontinuities is dependent on the number of rectangles in the Riemann approximation B. In this image, B = 10,000.

occur in practice. This pattern holds for all values of ε. This removes one possible obstacle for the usability of the CTA, as we now have a clear indication the output complexity will likely be a managable size.

Applying simplification does a lot to reduce the output complexity for a high enough δ. Simplifying the central trajectory gives a smaller output than simplifying the input trajectories and then applying the CTA. However, the difference is small, so storing the simplified input trajectories to save on memory can be a good option. Of course, simplification also destroys some of the data, so a good trade-off between memory footprint and data utility must be made based on the use case. Since we have shown that the output complexity does not become that large in the first place, simplification may not be needed.

Future work can be done in testing the algorithm in situations where n or τ are very large. This does pose some challenges, as it is hard to find publicly available data sets that contain many different very large clusters that are easily extracted and easily separable from noise such as stay points. Anonymization methods that have been applied to data sets may also make it harder to find very large clusters.

Chapter 5

Road Network Generalization

5.1 Introduction

Road network generalization is an *application task* that reduces a road network in size by selecting only the most relevant roads. There are many reasons for road network generalization: (i) visualization at a smaller scale than the data provided; (ii) efficiency of road network analysis; (iii) suitability for road network analysis. Research on this topic started in the 1990s by Thomson and Richardson [126] and Mackaness and Beard [104]. These early approaches assumed that a road network and a target map scale or detail level were given, and the objective was to select a suitable subset of the roads used for mapping. Later research went further and included road pattern preservation, generalization in the context of other map features, continuous generalization to support on-line zooming or scale-less databases, and inclusion of road usage data. The last aspect characterizes data-driven methods.

Data-driven methods rely on other data than just the road network. The most obvious other data source is car trajectories. In the last ten years the size of collections of vehicle trajectory data has grown tremendously, allowing a host of route planning and traffic analyses to be performed. Trajectories can first be preprocessed by map-matching so they can be treated as *routes* on the road network rather than paths through the plane. The measured times and locations are discarded and instead the route is stored as a list of used network edges. Since route planning and traffic analysis often use a road map as an interface, visualization and analysis of road networks are linked. Imagine a user selecting a start and a destination on a road map, along with specifications on the desired route. Then a route will be computed

and shown on the map. Ideally, the base map is not too cluttered due to showing all roads, but also, ideally the computed route uses roads which are already visible on the map. If we want these two features simultaneously, we can use *route-preserving road network generalization.*

We formalize route-preserving road network generalization as follows: Given a road network, represented by an embedded graph G, a collection R of routes on this network, and a length budget B, compute a subgraph of G whose summed edge length is at most B and which contains the maximum number of routes of R in full. Alternatively, we wish to compute the subgraph that has the maximum total length of routes fully contained.

We call these two versions *(full) route containment by number* and *(full) route containment by length.*

Let us examine this problem statement more closely and motivate it.

Simplicity First, we observe that the problem statement is simple and well-defined. Simplicity is an important feature. While a practical, useable implementation of road network generalization may need to take multiple, conflicting criteria into account by combining them, from the academic perspective it is important to understand to what extent a simple method can already solve the problem. Simple solutions that perform well have more predictable behavior and can be adapted to a truly well-functioning method more easily.

Parameters Second, we observe that the budget represents the degree of generalization. For example, our budget could be half the length of the original road network, leading to less road density and a better overview. The budget is an intuitive, useful input parameter.

Optimization criteria Third, we notice that the optimization to contain as many routes as possible corresponds directly to letting the subgraph represent as many driven routes as possible, with preference for routes that are driven more often. This is precisely what we aim to achieve. Optimizing the number of routes contained in the subgraph will give preference to including short routes, which is undesirable, and hence we expect that optimizing the total length of all completely contained routes gives better results.

Input assumptions Fourth, our problem of interest is the computation of the subgraph. We assume that road matching the input trajectory data has already been done. We assume that all routes lie on the full road network G. We also assume that all routes are single drives between an origin and a destination. We do not discuss the associated problem of route segmentation.

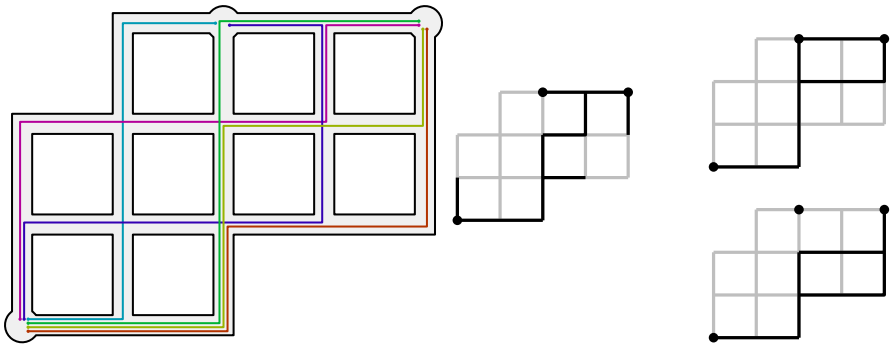


Figure 5.1: Left, a road network with six routes drawn. Middle, generalization according to most used roads with a budget of 11: all roads covered twice or more are chosen. Right, two equivalent outcomes of route-preserving generalization with a budget of 10: there are two ways to contain two routes with a budget of 10. A budget of 11 does not help to contain more routes.

Limitations Fifth, the solution to the problem is not necessarily a connected subgraph of G. Also, it does not lead to continuous generalization of the road network. These two desirable features can be incorporated without much effort, however, in a heuristic implementation.

Finally, we want to emphasize the subtle difference between maximizing the total number of routes (or their lengths) that are fully contained, and maximizing the total road usage of the roads chosen in the generalized network. In the latter problem, we can convert the routes into counts on the edges of the graph G, and otherwise forget about these routes. We believe that our version of preserving full routes gives rise to fewer artifacts. Figure 5.1 illustrates the difference and potential for artifacts on a small example. Maximal road usage leads to three "dead ends" in the generalized network that are not at destinations, and not a single route is fully preserved. Route-preserving generalization does not have dead ends, and two full routes are preserved. The example is not dependent on the chosen budget; other budgets give similar output.

▸ 5.1.1 Results

We first study the theory of route-preserving road network generalization (RPRNG). Like many interesting optimization problems, RPRNG is NP-hard. We show that this is already the case for very simple graphs, namely those without any cycles.

We also show NP-hardness for planar graphs if the routes hardly overlap: even if for every feature of the graph, at most two routes of R cover it, the problem is still NP-hard. On the other hand, the problem can be solved in polynomial time when two assumptions are made simultaneously: the graph has no cycles, and every graph feature is covered by at most a constant number of routes in R. See Table 5.1.

Since the problem is NP-hard for realistic input assumptions, a different approach is needed. To assess the idea of optimizing route covering, we developed and implemented heuristics to solve the problem. We incrementally choose the next route from the collection, based on a score function. When a route overlaps considerably with many other routes, it will receive a low score. We incrementally keep choosing the lowest scoring routes until the budget is used up. There are two options: either we compute scores from the start and keep on using these scores, or we adapt the scores after every incremental choice. We can expect that the former method is more efficient and the latter method gives better optimization.

Both of these incremental methods yield a *continuous* method for road network generalization; that is, they can be used in maps that support interactive zooming. In fact, the NP-hardness of the fixed target-scale problem is closely related to the fact the optimal solution to the problem is not continuous as the budget is increased or decreased, and pieces of the input graph may appear and disappear during zooming in one direction.

We study the outcome of four data-driven road network generalization methods. The first method performs generalization based on road usage, without attempting to include full routes (chosen as a baseline). The other three are route-preserving.

Our analysis is both quantitative and qualitative. Quantitatively, we report the number of full routes preserved and their summed length, the average coverage (ply) of chosen edges, the number of connected components, and the number of road network leaves that are not at origins or destinations of routes. Qualitatively, we inspect the resulting networks to see where they are different and where they are the same, and make observations. For example, we observe that the baseline has many more leaves not at origins or destinations, but RPRNG still has some, when a U-turn was made.

▸ 5.1.2 Related work

Many papers have been written on road network generalization; this chapter gives an overview of a selection only.

Among the criteria used to select roads, many are cartography-focused, in the sense that the generalized road network should “look good”. These criteria include

avoiding small faces between the roads, avoiding coalescence, and controlling density [43, 51, 78, 99, 103, 91, 136, 143]. Other methods give preference to sequences of roads that are smooth continuations of each other (strokes) [23, 101, 127, 125]. A global criterion that is often used is connectivity of the generalized network [44, 104], which is one of several structural graph-based criteria [76, 77, 131].

The most important criterion for continuous generalization is avoiding sudden changes, and avoiding that when zooming in one direction, a road segment disappears and then reappears [44, 123]. This type of behavior is to be avoided for all continuous generalization operations. One of the existing methods to road network generalization is coined "selective omission", where road segments, extended by good continuation to strokes, are scored based on geometric, topological, and attribute factors [43, 143]. As mentioned, our method can be adapted to support such continuous zooming, since it can be seen as a new way to perform selective omission.

One of the first road network generalization methods can be called a precursor to data-driven generalization. Thomson and Richardson [126] let a subset of the vertices of a road network be sources and destinations, and they compute shortest paths between each pair. This gives artificial road usage, and they select the most used roads according to this computation. Similarly, road usage can be estimated based on an agent-based simulation [110]. While data-driven geography [108] has been around for longer, data-driven road network generalization has —to the best of our knowledge— not been studied until very recently. Fekete et al. [54] focus on finding the optimal placement for k points on a road network to maximize the length of real-data subtrajectories captured between each pair of points, which could also be applied for road network generalization. Yu et al. [132] refine road network generalization based on selecting strokes in the network by including traffic flows mined from trajectory data.

Besides the few other data-driven road network generalization papers, many methods use a scoring of roads, and clearly the road score can be based on actual road usage, which would be data driven (in both meanings of the word "driven"). Note that our route-preserving approach uses data in a different way: it is driven-route driven.

Our research is related to road network construction from trajectories, a topic on which a lot of research exists (e.g., [11, 30, 81, 122]). The emphasis of that line of research is different, since the challenge is dealing with noise and deciding when trajectories follow the same (unknown) road. Our research is also related to the problem of hotspot computation on road networks based on trajectory data [34, 97], which concerns another application with a data-driven solution, based on the same

input data.

From the theoretical perspective, our research problem involves a weighted graph with paths on that graph. Our problem does not use coordinates of the road network explicitly (only to determine lengths of road segments), nor the coordinates and times of the trajectories. Hence, the abstract view of our problem is a graph problem and not a network problem. Since graph problems typically do not come with a set of paths on that graph, there are no closely related graph problems. However, there are some connections which allow us to prove NP-hardness of route-preserving road network generalization.

▸ 5.2 Theoretical results

Table 5.1: Algorithmic and hardness results for graph classes of G and bounds on the ply.

	path	tree	planar graph
ply 1	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
ply 2	$O(n\|R\|^2)$	$O(n\|R\|^2)$	NP-hard
ply c	$O(n\|R\|^2 c)$	$O(n\|R\|^2 2^{2c})$	NP-hard
ply ∞	$O(n\|R\|^3)$	NP-hard	NP-hard

In this section we show that both versions of the RPRNG problem are NP-hard, even for simple and sparsely covered road networks. We present polynomial-time algorithms for two special cases of the number of routes RPRNG problem, when the road network is a path (with arbitrarily heavy coverage), and when it is a tree (with each segment of the network covered by at most c routes, for some fixed constant c).

Let $G = (V, E)$ be the graph representing the road network, with vertices V and edges E, and let R be the set of routes in G. For simplicity of presentation, we assume that each route in R begins and ends at a vertex in V, and that the routes are simple (i.e., non-self-intersecting). Note that it is straightforward to extend our algorithm to the case of non-simple routes with minor changes to the dynamic programming formulation. Furthermore, assume that each edge in G is traversed by at least one

route, otherwise we remove such edges in a preprocessing step. Define the *ply* of an edge $e \in E$ to be the number of routes in R traversing e. Similarly, the *ply* of a vertex $v \in V$ is the number of routes in R traversing v.

The route length version of the RPRNG problem is NP-hard by a very easy reduction from the Subset Sum problem even for the case when the routes are pairwise disjoint and G is a path.

Theorem 5.2.1. *The RPRNG problem with the objective to optimize the total length of covered routes is NP-hard even if the maximum ply is* 1 *and the graph is a path.*

The number of routes version of the RPRNG problem is the easier version, and allows for polynomial-time solutions for some special cases. Our theoretical results for this version of the problem are summarized in Table 5.1. We present two polynomial-time algorithms for a dual formulation of the RPRNG problem for the cases when G is a path with edges and vertices of unbounded ply, and when G is a tree with maximum ply bounded by some constant c. In this dual formulation we are given a number of routes to be covered, and the goal is to minimize the total length of the resulting subgraph. Having a polynomial-time algorithm for this formulation, we can solve the original RPRNG problem by performing a binary search on the number of routes to be covered, resulting in an algorithm with an extra logarithmic factor in its running time.

We then present two NP-hardness proofs for the RPRNG problem in the cases when G is a tree with unbounded ply, and when G is a planar graph with max ply at most 2. We also show that this last version is still NP-hard even if we add the additional restriction that all routes are required to be shortest paths on the graph. Note that the case when G is a graph with max ply 1 is trivial. Indeed, in that case G reduces to a set of disconnected paths, and a simple greedy strategy can be applied to solve the problem.

▸ 5.2.1 When G is a path

We start with a dynamic programming algorithm for the case when $G = (V, E)$ is a path; the maximum ply may be arbitrarily large. Let $V = \{v_1, v_2, \dots, v_n\}$ be n vertices of G embedded on a line from left to right, and let edges $E = \{e_i = (v_i, v_{i+1}) \mid 1 \le i < n\}$ (refer to Figure 5.2). Let R be the set of routes in G, and let R_i be the set of routes that each starts on or before v_i and ends after v_i. That is, for each $r \in R_i$ we have that r starts at some vertex v_j with $j \le i$ and ends at some vertex v_k with $k > i$.

Let $F(i, k, S)$, where $i \le n$, $k \le m$, and $S \subset R_i$, denote a subproblem in our dynamic programming formulation on the first i vertices, with k routes to be covered, and

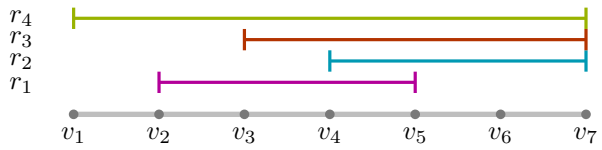


Figure 5.2: Example of the RPRNG problem on a path graph with seven vertices and four routes. An optimal solution to cover two routes is a path between v_3 and v_7, which covers routes r_2 and r_3.

such that the routes in S are all covered up to vertex v_i. The goal is to minimize the total length of an output subgraph in $F(i, k, S)$. A naive dynamic programming recursion is given by the following formula:

$$F(i, k, S) = \min_{S \cap R_{i-1} \subseteq S' \subseteq R_{i-1}} \left[F(i-1, k - |S' \setminus R_i|, S') + cost(e_{i-1}, S') \right] \tag{5.1}$$

With boundary conditions $F(i, k, S) = 0$ if $i = 0$ and $k \leq 0$, and $F(i, k, S) = \infty$ if $i = 0$ and $k > 0$. The cost function $cost(e_{i-1}, S')$ depends on the choice whether or not to include the last edge $e_{i-1} = (v_{i-1}, v_i)$ into the output graph G'. Note that this choice is dictated by the subset S'. If there are any routes in the final solution that cover e_{i-1}, then they must be included in S'. As all the routes in S' use the edge e_{i-1}, it must be included into the output graph if S' is non-empty. Therefore, we define the cost function as follows:

$$cost(e, S') = \begin{cases} 0\,, & \text{if } S' = \emptyset\,, \\ \|e\|\,, & \text{otherwise}\,. \end{cases} \tag{5.2}$$

Note that for a given i, the above recursion considers exponentially many subsets S'. To optimize this, we can instead restrict our dynamic programming to consider only maximal subsets S'. That is, let $r \in S'$ be a route with the left-most starting vertex v_j. Then, we can without additional cost include in S' all routes r' which start between vertices v_j and v_i and end after vertex v_i. Thus, as the size of S' is bounded by $|R|$, instead of considering $O(2^{|R|})$ possible subsets, we can restrict ourselves to only $O(|R|)$ subsets. We obtain a dynamic programming table of size $n \times |R| \times O(|R|)$, where n is the number of vertices in G, and $|R|$ is the number of routes. For each subproblem we spend $O(|R|)$ time, giving us a polynomial-time algorithm. Thus we conclude with the following theorem.

Theorem 5.2.2. *The RPRNG problem with the objective to maximize the number of covered routes can be solved in $O(n|R|^3)$ time when G is a path, where n is the number*

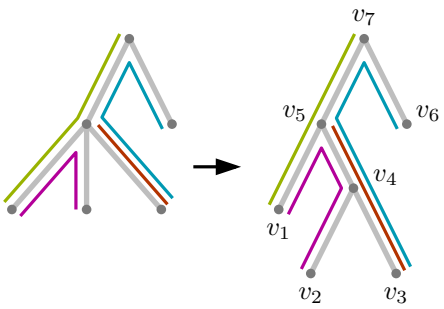


Figure 5.3: Example of the RPRNG problem on a tree graph with four routes. Vertices of degree higher than three are split to form a binary tree, weight of edge (v_4, v_5) is set to 0. There are multiple optimal solutions to cover two routes, for example, a subtree rooted at vertex v_5.

of vertices in G, and $|R|$ is the number of routes.

▸ 5.2.2 When G is a tree with bounded ply

We now modify our dynamic programming algorithm from the previous section to solve the RPRNG problem in the case when G is a tree, and the maximum ply of edges and vertices is bounded by some constant c. The vertices of degree higher than three can be split by inserting zero-weight edges to form a binary tree. Choose an arbitrary root, and order the vertices according to the post-order traversal (refer to Figure 5.3). Thus, the leaves have smaller indices than their parents. Let R_i be the set of routes which begin at some vertices in the subtree rooted at v_i, and end at some vertices outside of the subtree. That is, R_i is the set of routes which start on or below v_i and traverse v_i from bottom to top.

Consider a subproblem $F(i, k, S)$ on a subtree rooted at the vertex v_i, where at least k routes must be covered, including the set $S \subset R_i$ of routes traversing v_i from bottom to top. In our dynamic program, we will merge the solutions of the subproblems defined on the subtrees rooted at the children of v_i. Similarly to the previous section, the dynamic programming recursion is given by the following formula:

$$F(i,k,S) = \min_{\substack{S\cap R_\ell \subseteq S_\ell \subseteq R_\ell \\ S\cap R_r \subseteq S_r \subseteq R_r \\ k_\ell + k_r = k - |S_\ell \cap S_r|}} \begin{bmatrix} F(i_\ell, k_\ell, S_\ell) + cost(e_{i_\ell}, S_\ell) + \\ \\ F(i_r, k_r, S_r) + cost(e_{i_r}, S_r) \end{bmatrix} \tag{5.3}$$

Where i_ℓ and i_r index the left and right child of the vertex v_i respectively, and e_{i_ℓ} and e_{i_r} denote the edges (v_{i_ℓ}, v_i) and (v_{i_r}, v_i). The boundary conditions are $F(i, k, S) = 0$ if v_i is a leaf and $k \leq 0$, and $F(i, k, S) = \infty$ if v_i is a leaf and $k > 0$. The values k_ℓ and k_r in the two subproblems depend on the value k and the number of routes covered by the solutions of the two subproblems which are traversing v_i from the left subtree into the right subtree (for example, for the vertex v_5 in the Figure 5.3, the red route is such route). More specifically, let k' be the number of routes in $S_\ell \cap S_r$, then $k = k' + k_\ell + k_r$.

Unlike in the previous section, we no longer can consider only maximal subsets S of the routes traversing v_i, thus we need to consider all possible subsets. Since the maximum ply is bounded by c, the number of subsets of routes traversing v_i is 2^c. The size of the dynamic programming table becomes $n \times |R| \times 2^c = O(n|R|)$, and we spend $O(|R|2^c) = O(|R|)$ time per subproblem. We conclude with the following theorem.

Theorem 5.2.3. *The RPRNG problem with the objective to maximize the number of covered routes, when G is a tree and the maximum ply is bounded by some constant c, can be solved in $O(n|R|^2 2^{2c})$ time, where n is the number of vertices in G, and $|R|$ is the number of routes.*

▸ 5.2.3 When G is a tree with unbounded ply

In this section we show that the RPRNG problem is NP-hard by a reduction from the clique problem, which asks whether there exists a clique of a certain size in a given graph. Given an instance of the clique problem on a graph G' with sought clique of size k', we construct an instance of the decision version of the RPRNG problem consisting of a road network graph G, a set of routes R, an integer budget B on the total length of the output network, and an integer k—the number of routes to be covered. We show that there is a clique of size at least k' in G' if and only if there is a subgraph of G of total length at most B which completely covers at least k routes from R.

Specifically, let graph G' have n vertices $\{v_1, v_2, \ldots, v_n\}$. We construct G to be a star graph with $n + 1$ vertices $\{u_0, u_1, \ldots, u_n\}$, with edges (u_0, u_i) for all $1 \leq i \leq n$ (refer to Figure 5.4), such that each edge has length 1. For all edges (v_i, v_j) in G', we add a route to R consisting of two edges (u_i, u_0) and (u_0, u_j). Finally, we set $B = k'$ and $k = k'(k' - 1)/2$.

If there is a clique of size k' in G', then there is a set of B edges in G whose union covers k routes in R. Indeed, let $\{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$ be the vertices in a clique in G', each edge connecting a pair of vertices in the clique has a corresponding route in R.

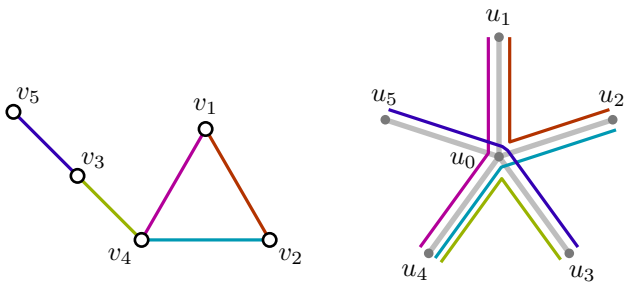


Figure 5.4: Left: graph G' from a clique problem instance. Right: graph G with five routes corresponding to the edges of G'. Subgraph of G on vertices $\{u_0, u_1, u_2, u_4\}$ corresponds to the clique $\{v_1, v_2, v_4\}$ in G'.

Thus, the subgraph of G on vertices $\{u_0, u_{i_1}, u_{i_2}, \dots, u_{i_k}\}$ has $B = k'$ edges and covers the $k = k'(k'-1)/2$ routes in R corresponding to the edges of the clique.

Conversely, if there is a subgraph of G consisting of $B = k'$ edges that covers $k = k'(k'-1)/2$ routes in R, then there are k' vertices in G', each pair of which is connected by an edge corresponding to one of the covered routes in R.

Therefore, we conclude with the following theorem.

Theorem 5.2.4. *The RPRNG problem with the objective to maximize the number of covered routes is NP-hard when G is a tree and the maximum ply is unbounded.*

▸ 5.2.4 When G is a planar graph with bounded ply

We will now show that the RPRNG problem is NP-hard when G is a planar graph even if the ply on edges and vertices is bounded by 2. We again reduce from the clique problem.

Consider an instance of the clique problem on a graph $G' = (V', E')$, with an integer k. We construct an instance of the RPRNG problem consisting of a graph G, a set of routes R, a budget B, and an integer k, such that G' has a clique of size k if and only if there exists a subgraph of G of total weight at most B which covers at least k routes.

To create this instance, we set G to be a $|V'| \times 2|E'|$ grid graph, where we assign one row to each vertex of V'. We assign two adjacent columns of vertices to each edge of E', and set the weight of the $|V'|$ horizontal edges connecting these adjacent columns to 1. All other edges have weight 0. We create a route for each vertex of V' that lies on the row associated with its vertex, except for the edges in columns

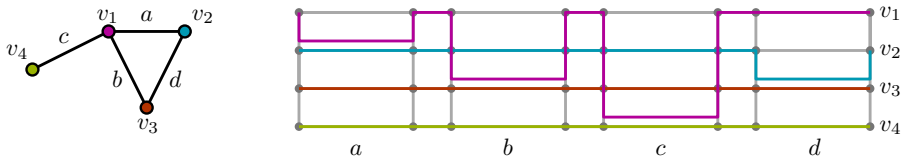


Figure 5.5: Left: graph G' of a clique problem instance. Right: corresponding grid graph G, and four routes corresponding to the vertices of G'. Edges in a column associated with an edge of G' have weight 1, all other edges have weight 0.

associated with edges where the associated vertex is the lower-indexed endpoint. For those edges, the route travels vertically to the row associated with the higher-indexed endpoint and uses that edge instead before vertically going back to its own row. See Figure 5.5. We set $B = k|E| - k(k-1)/2$. It becomes clear that this is a correct reduction when we consider that if we pick k routes that do not overlap, we need a budget of $k|E|$. If two selected routes overlap, the amount of budget needed decreases by 1. If the savings sum to a total of $k(k-1)/2$, this means we can select k vertices that share $k(k-1)/2$ edges between them in G', implying they are a clique of size k. It is easy to see that in the grid graph, no edge or vertex is covered by more than two routes.

Theorem 5.2.5. *The RPRNG problem with the objective to maximize the number of covered routes is NP-hard even if G is a planar graph and the maximum ply is bounded by 2.*

Note that in the above construction we can set the weight of the 0-weight edges to be a small positive value $\varepsilon \ll 1$, and thus obtain the same result for a graph with non-zero edge weights.

▸ 5.2.5 When G is a planar graph, ply is bounded, and routes must be shortest paths

Another special case we will investigate is like the previous case, but we also require that each route is a shortest path on G. This is a natural extension as real world routes tend to be close to the shortest path as drivers want to reach their destination as fast as possible. However, we will show that this additional restriction does not prevent the problem from being NP-hard.

Our reduction is once again from the clique problem. Consider an instance of the clique problem on a graph $G' = (V', E')$, with an integer k'. We construct an instance of the RPRNG problem consisting of a graph G, a set of routes R, a budget

B, and an integer k, such that G' has a clique of size k' if and only if there exists a subgraph of G of total weight at most B which covers at least k routes. Our instance will be constructed such that G is a planar graph, and R will only consist of shortest paths where the maximum ply of any edge is 2.

First we will construct the graph G for our instance. We will do this by first creating an arrangement of shapes which we will then transform into a graph by placing vertices on the shapes and connecting them with edges. The number of vertices we will be placing on each shape will be a function of a fixed number n. Let $n = \max\{|V'|, k^2\}$.

For each vertex of G' we create a circle. If two vertices of G' share an edge, their circles are connected by a straight line segment, which we will call the edge's arc. To ensure that there are no intersections between arcs and circles that are not incident to the arc, we place the circles on the corners of a regular $|V'|$-gon.

Now we replace each arc with a chain of edges: At the places where two arcs intersect a vertex is placed, and additional vertices are placed such that each arc is divided into exactly n^2 edges. These additional vertices are placed only on the parts of the arc between a circle and an arc intersection, not in between two arc intersections. The endpoints of the edge chain lying on an arc are vertices which lie on one of the circles corresponding to the vertices of G' incident to the arc's edge. Additional vertices are placed on the circles such that each circle has $2n^3$ vertices on it. Each vertex is connected by edges to its adjacent vertices on the circle, such that a cycle of $2n^3$ edges is created lying on the circle. The weight of each edge is set to 1. This finishes our construction of G.

Now we will create the set of routes R. For each cycle, we will create a set of $2n$ routes lying on it. We start by creating a route r_1 covering $n^2 + n$ edges somewhere on the cycle. Then, we create a second route r_2 of $n^2 + n$ edges on the cycle, such that the last n edges of r_1 overlap with the first n edges of r_2. We can keep creating routes like this until we have covered the entire cycle. The last route we create, r_{2n}, has its last n edges overlap with the first n edges of r_1. The cycle then has $2n$ routes on it of $n^2 + n$ edges each, where the first and last n edges overlap. This results in $2n^2$ edges with ply 2 and ply 1 everywhere else on the cycle. We can always construct the cycle and routes in such a way that each cycle vertex incident to an arc and the next n edges on the cycle have ply 1.

Besides these $2n^2$ routes that lie fully on cycles, we also create one route for each arc. These routes cover every edge of an arc, as well as n edges of each cycle the arc is incident to. The addition of these routes completes our set R. See Figure 5.6 for a sketch.

We let $B = (2k'n + k'(k' - 1)/2)n^2$ and $k = 2k'n + k'(k' - 1)/2$. We will know G'

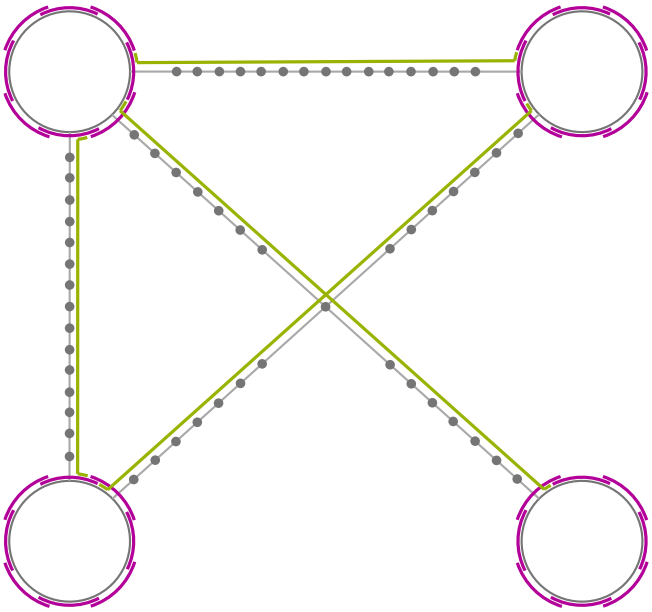


Figure 5.6: Reduction from the clique instance shown in Figure 5.5. The constructed graph consists of cycles of $2n^3$ vertices on an n-gon connected by arcs. Points are added to the arcs so that each arc consists of n^2 edges. Purple routes cover $n^2 + n$ edges, green routes cover $n^2 + 2n$ edges. Only by selecting the routes associated with edges and vertices that form a clique can we get enough routes using our budget.

has a clique of size k' if and only if we can capture k routes of R using at most B budget. To see that this reduction works, consider that the routes that lie fully on cycles have length $n^2 + n$ and the routes lying on arcs have length $n^2 + 2n$. Capturing $2k'n + k'(k' - 1)/2$ routes with the budget results in an average budget per route of n^2. Since each route consists of more than n^2 edges, our captured routes must have significant overlap. If we pick all of the edges on a single cycle we capture $2n$ routes, spending n^2 budget per captured route. Doing this for k' cycles gives us $2k'n$ captured routes with $n^2k'(k' - 1)/2$ budget left over, which is not enough to pick all of the edges on an additional cycle. However, if the edges of both cycles incident to a single arc have already been chosen, we only need n^2 additional budget to cover the rest of the edges on that arc. Each arc we can cover like this captures an additional route. So if there are $k'(k' - 1)/2$ of such arcs, i.e. the k' vertices associated with our chosen cycles form a clique, we can capture enough routes without exceeding the

budget. Any other way of capturing k routes will not be able to get enough overlap between routes and so too much budget would be needed.

It is clear that the transformation is polynomial, and that the constructed graph is planar, has a maximum ply of 2, and only has shortest paths as routes. This gives us the following theorem.

Theorem 5.2.6. *The RPRNG problem, with the objective to maximize the number of covered routes is NP-hard even if G is a planar graph, the maximum ply is bounded by 2, and all routes are a shortest path.*

▸ 5.3 Experiments

To assess and validate the concept of route-preserving road network generalization, we implemented three simple heuristics and a baseline density-based method, and compared their performance on GH and NSH, two sets of vehicle routes from the South-West of the Netherlands. See Appendix A for details on the data sets.

▸ 5.3.1 Data

As input to our problem, we need two types of data: a road network, and a set of routes on the network.

As our road network, we use the freely-available OpenStreetMap (OSM) [115]. To prepare the map for our experiments, we clipped the map to two rectangular sections around our areas of interest, and further restricted the road network to those sections that are actually used by at least one of the routes. Finally, we retain only the largest connected component of the remaining network in each area. Figure 5.7 shows a small relevant section of the map. As our routes, we use NSH and GH. Since routes are not provided as paths in the graph but raw GPS coordinates, we first run them through a map-matching algorithm. We used the GraphHopper system [64], which is based on the hidden-Markov method by Newson and Krum [112]. In addition, we performed some data-preprocessing to make sure all routes are completely within the area of interest, are of sufficient length, and start at vertices. A full description of the data preparation pipeline can be found in Appendix C, specifications for the final data sets can be found in Table C.1.

▸ 5.3.2 Heuristics

In order to obtain reasonable results with simple methods, we propose several heuristics. Our heuristics are based on selecting sections of the road network that are

Figure 5.7: Our data set from Leiden (NSH). Top: OSM network. Bottom: The network restricted to roads used in the data set.

heavily used. Formally, if G is the road network and R is the set of routes, then we define the $ply(e)$ of an edge e of G to be the total number of routes in R which contain e.

The *weight* of an edge, $w(e)$, in this section always refers to its Euclidean length. We also define the *length* of a route to be the total sum of the weights of edges that appear in the route:

$$\ell(r) = \sum_{e \in r} w(e) \tag{5.4}$$

Note that, for the purpose of these heuristics, a route is simply a set of edges of G; in particular, even if a route uses the same edge multiple times, it appears only once in the set. This is justified by our goal of extracting a portion of the network that is used by many *distinct* routes.

All heuristics are greedy in some fashion and attempt to select edges with high

ply, since these are used by many different routes. Hence, the heuristics are not tailored towards a specific goal, and their performance will be evaluated for both versions of the problem (route containment by number or route containment by length).

`baseline` First, we describe a baseline heuristic, which is data-driven but not route-preserving, to compare the performance of our route-preserving heuristics against.

We simply sort the edges of G by decreasing value of $ply(e)$. Then, we greedily select edges until we have selected a total length of B. We consider edges of G to be atomic; that is, we do not choose edges partially. If the remaining budget is not enough to pick the full length of the highest ply edge that is left, instead we pick the next-highest ply edge that still fits in the remaining budget.

`score` Our simplest route-preserving heuristic is based on the following simple idea: routes that overlap with many other routes will, on average, consume less budget.

For this heuristic, we assign to each route $r \in R$ a *score*, $s(r)$, which is defined as follows: for each edge of r, we count how many routes use it (including r itself), and inversely weigh the edge by this number:

$$s(r) = \frac{1}{\ell(r)} \sum_{e \in r} \frac{w(e)}{ply(e)} \tag{5.5}$$

In other words, for sections that are covered only once, the score is the normalized length. For sections covered once more, we take the length /2, for sections covered twice more, take the length /3, and so on, and add up these weighted lengths.

Then, we sort all routes by increasing score, and we simply greedily select routes (when we select a route, we take all edges of the route that were not selected yet, and include them in our output network), until we reach our budget. If we cannot select the next route in the set because we do not have enough budget, we skip it, and take the first (lowest-scoring) route for which we do still have enough budget.

`rescore` In our second route-preserving heuristic, we use the same score-based approach as for `score`. However, we now only select the lowest-scoring route, and recompute the scores of the remaining routes before selecting the next one. The rationale is that for edges which have already been selected, it no longer matters how many other routes use it, it is now "free".

Formally, when a subset $E' \subseteq E$ has been already selected, we define the score of a route $r \in R$ as the weighted sum over only those edges that have not been selected yet:

$$s(r) = \frac{1}{\ell(r)} \sum_{e \in r, e \notin E'} \frac{w(e)}{ply(e)} \tag{5.6}$$

The heuristic now evaluates the scores of all routes, selects the lowest-scoring route, re-evaluates the scores, again selects the lowest-scoring route, and so on, until we reach our budget.

`alternate` Finally, as our third route-preserving heuristic, we not only select the lowest-scoring route at each step, but we also *discard* the *highest-scoring* route. As with `rescore`, we re-evaluate the scores after each step. However, since we have now discarded potential routes from being selected later, we not only have to recompute the scores $s(r)$, but also $ply(e)$ for each edge.

Formally, when a subset $E' \subseteq E$ has been already selected and a subset $R' \subseteq R$ is still available, we redefine $ply(e)$ as the total number of routes in R' which contain e, and then redefine $s(r)$ as in Equation (5.6).

▸ 5.3.3 Experimental set-up

For our experiments, we prepared the data as described above and stored the resulting preprocessed networks and route collections, and we implemented the four heuristics in C++.

In our experiments, for each of the two data sets we run each of the four heuristics for 9 different budget values, corresponding to 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45%, and 50% of the total length of the input network, for a total of 36 runs. For each run, we collect the following information:

- The total number of routes fully covered.
- The total length of all routes fully covered.
- The total length of road network chosen.
- The average ply of the chosen portion of the network.
- The number of connected components in the output network.
- The number of degree 1 vertices (leaves) in the output network.

In addition, for each run we store the resulting network in its entirety for visual inspection.

▸ 5.3.4 Results

Quantitative results Now we will report on our collected statistics. The quantitative results for NSH are given in Figure 5.8, and those for GH are given in Figure 5.9.

We can see that for both data sets, the quantitative results are very similar. One difference is that GH has significantly more connected components and leaf vertices than NSH when the `baseline` heuristic is used.

For some of the measured statistics, the results for each method are nearly identical. For the average ply and the length of the selected subgraph, all methods perform the same, except for the `alternate` heuristic. For the number of connected components, all of the score-based heuristics perform the same, and only the `baseline` heuristic differs.

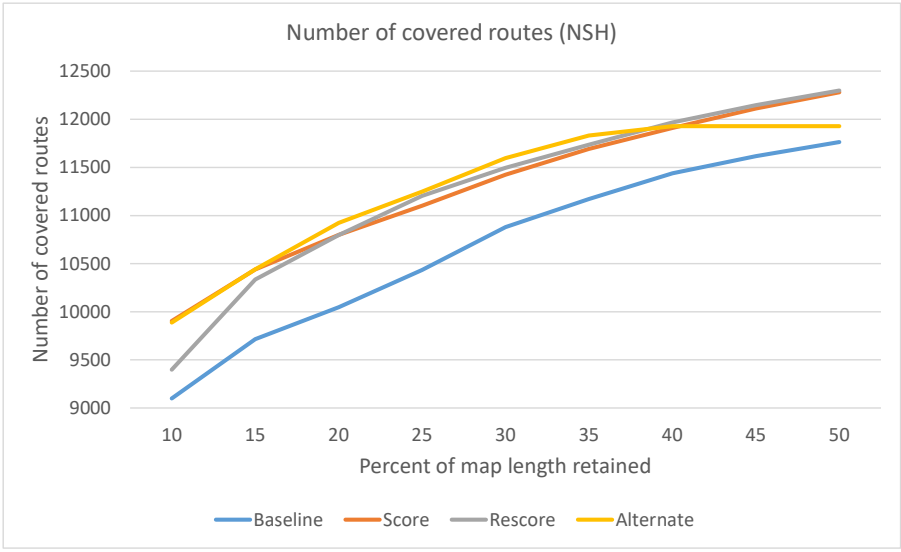
The total length of the output network is always very close to the target budget, except for the `alternate` heuristic when the target budget is 40% or higher, in which case the output network is significantly lower than the budget.

Qualitative results We show a selection of output road networks for visual comparison. To compare the different heuristics, in Figure 5.10 the resulting networks on the NSH data set, for a budget of 30% of the input length are shown.

To compare the effect of the budget, in Figure 5.11, we show the resulting network of the `rescore` heuristic on the same section of the NSH data set, for budgets of 10%, 30%, 50% of the original network. We notice that the smaller networks are always subnetworks of the larger ones, as desired for continuous zooming.

▸ 5.3.5 Discussion

As expected, our route-preserving heuristics appear to be better at preserving routes than the `baseline` heuristic. This is true regardless of whether we count the number of preserved routes or the total length. It is also apparent from the fact that the baseline heuristic produces maps with many small isolated components. This is also clear from visual inspection of Figure 5.10 (top left). The GH data set gives rise to more of these connected components (also leading to more leaf vertices). This is likely due the increased size of GH and its distribution of routes. Even so, the `baseline` heuristic still preserves a sizable number of routes even though it does not consider routes when making selections. The heuristic is helped by the fact that



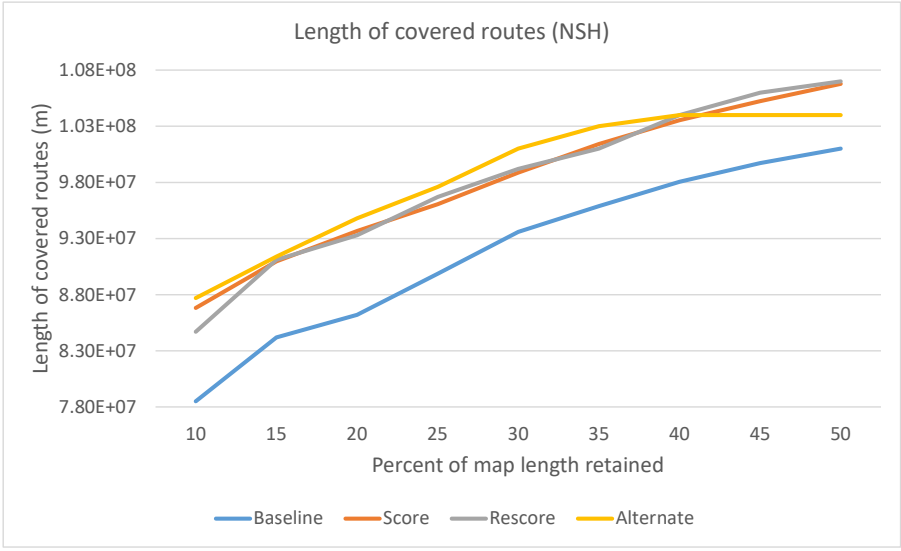
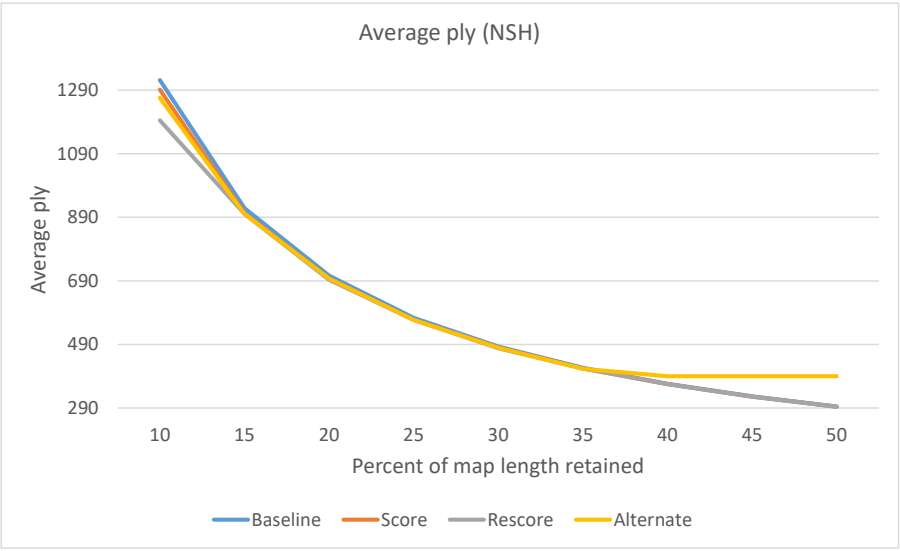


Figure 5.8: Quantitative results for NSH. Continued on next page.



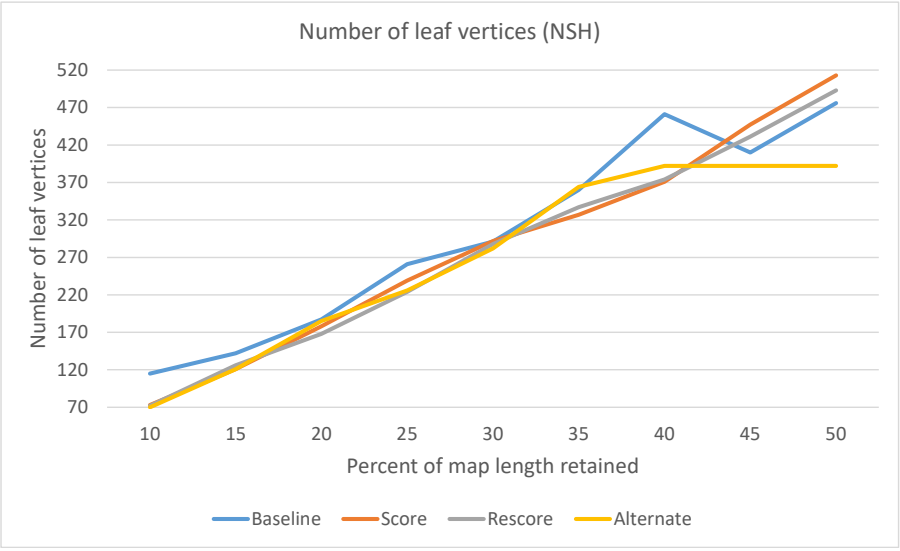
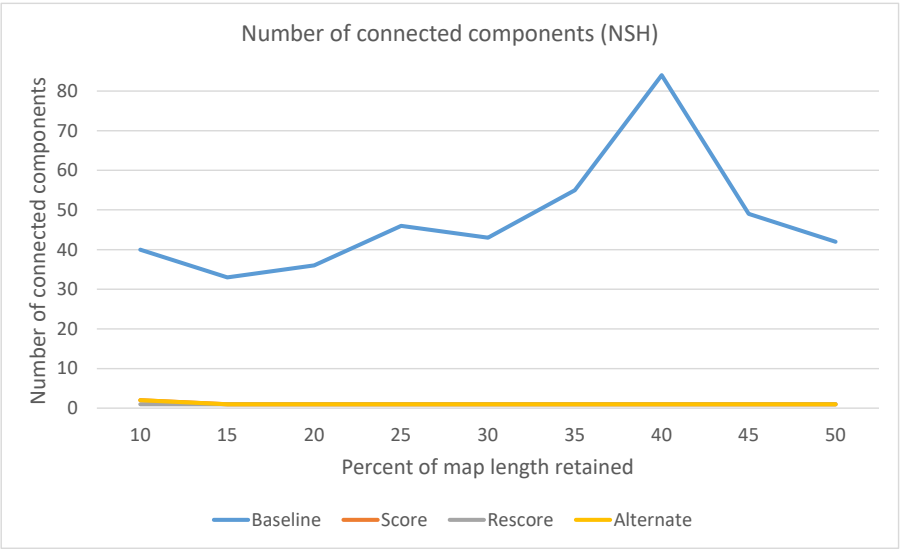


Figure 5.8: Quantitative results for NSH. Continued on next page.



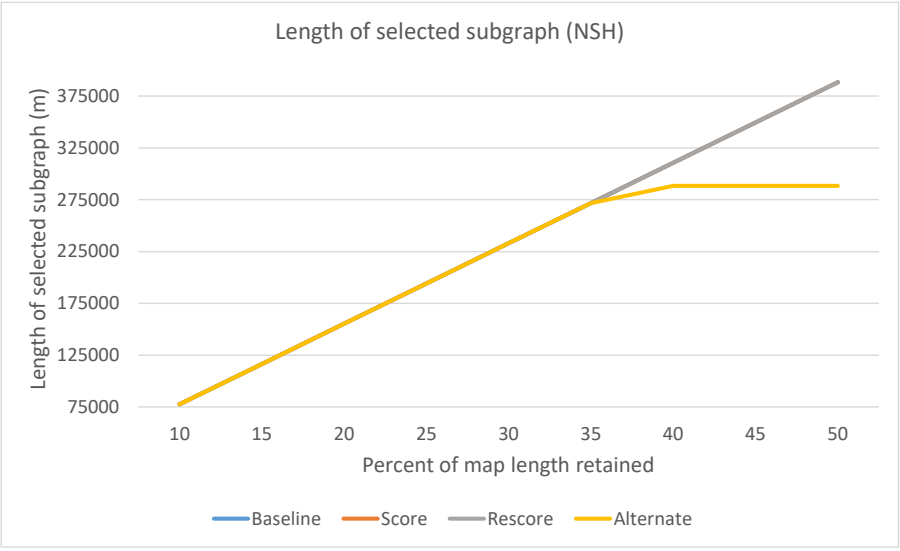
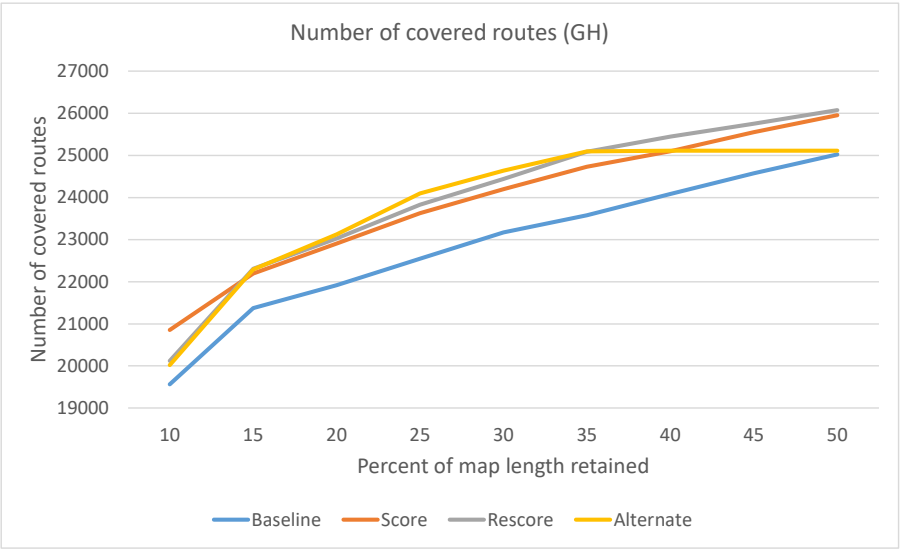


Figure 5.8: Quantitative results for NSH.



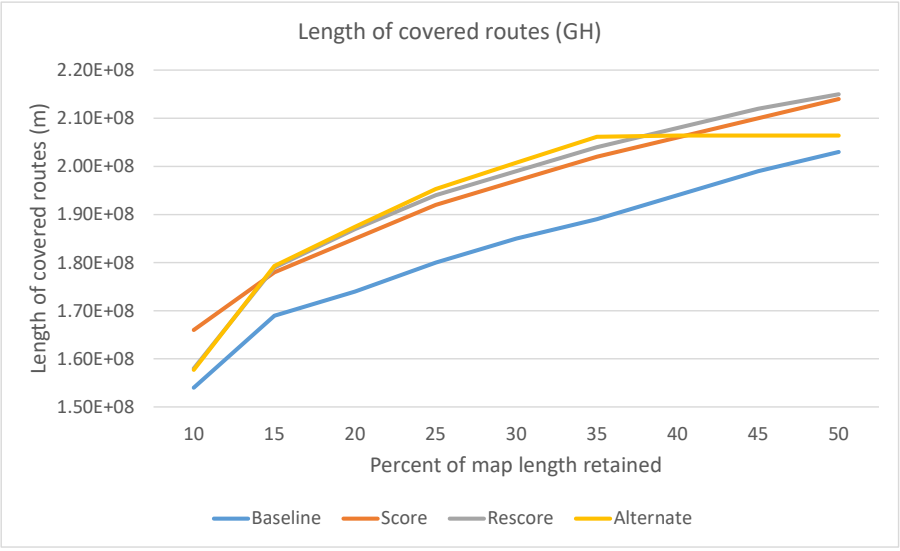
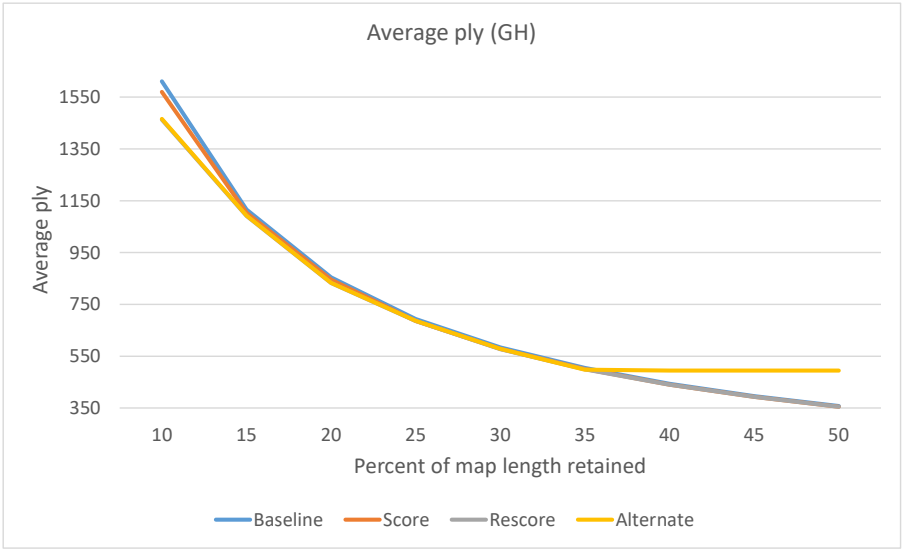


Figure 5.9: Quantitative results for GH. Continued on next page.



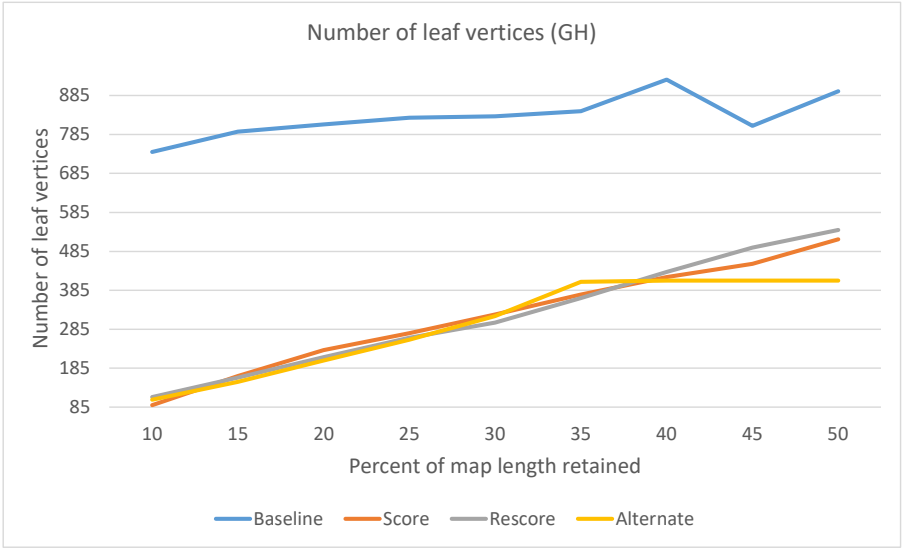
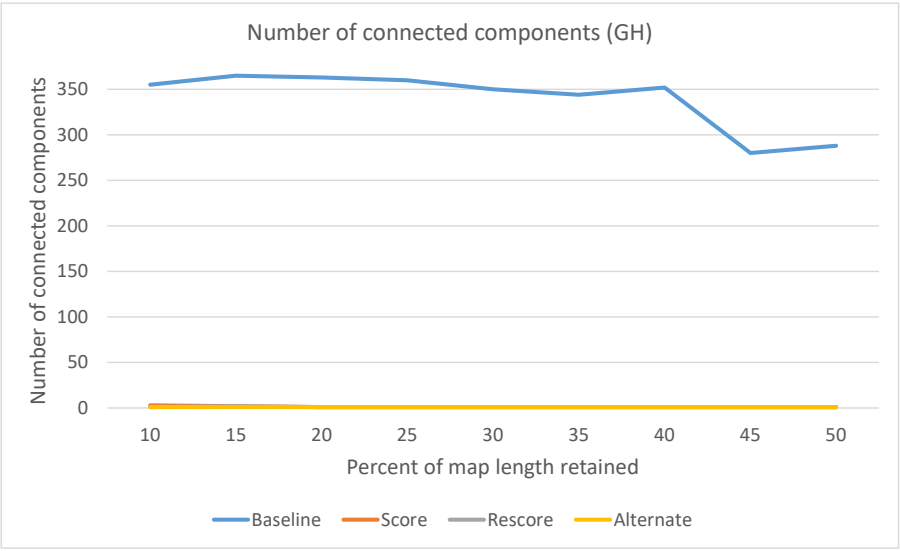


Figure 5.9: Quantitative results for GH. Continued on next page.



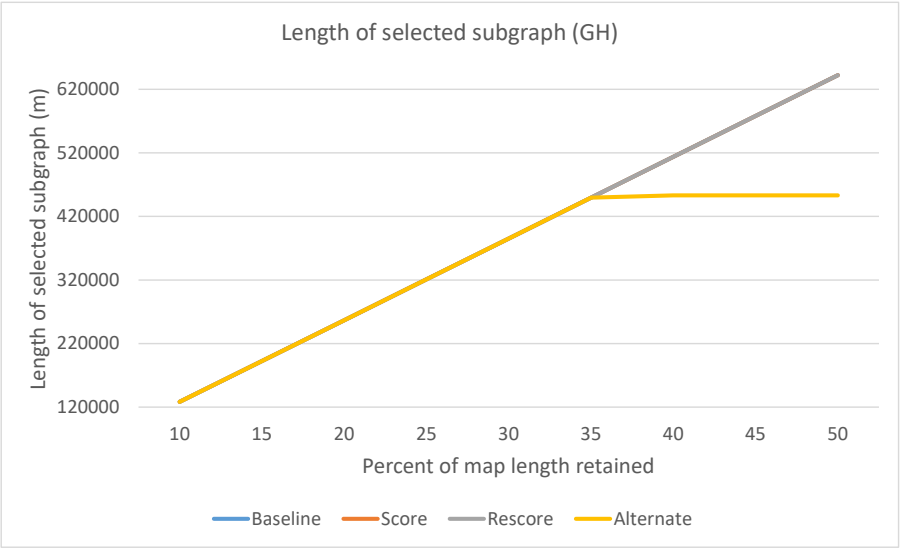


Figure 5.9: Quantitative results for GH.

Figure 5.10: The network restricted to 30% of the total road length under different heuristics. From the top down: `baseline`, `score`, `rescore`, `alternate`.

Figure 5.11: NSH under after application of the `rescore` heuristic with different budgets. The top image is the input graph, from the top down, the images correspond to a length budget of 50%, 30% and 10% of the input graph length, respectively.

routes lying on an edge with high ply are likely to also have high ply on their other edges, making it more likely that all the edges get selected.

It is less clear which of the route-preserving heuristics performs better. From Figures 5.8 and 5.9, it appears that the `alternate` heuristic is slightly better for low budgets, but the performance stabilizes once the target budget reaches about 35%. This appears to indicate that discarding non-promising routes is beneficial, but for higher budgets, this heuristic discards routes too easily and too often and eventually runs out of routes to include before the budget is used up. It would be interesting to investigate the balance between the budget and the condition for discarding routes from consideration.

Performance between the `score` and `rescore` heuristics is very similar. This suggests the extra computational cost for the `rescore` heuristic may not be justified. On the other hand, rescoring is a necessary basis for more sophisticated heuristics, such as `alternate`.

We observe that all four methods exhibit 'good continuation' to some extent: visually the maps are decomposable into long continuous strokes, rather than reading as many individual smaller strokes.

We do not observe much difference in the performance of the heuristics, nor in the general values of our statistics, between the two different data sets. The main notable difference between the four methods is the number of leaf vertices in the output of the `baseline` heuristic.

Visually, the outputs for the heuristics are somewhat similar. The bigger roads are present in the outputs of all the heuristics, but the results show more variance when it comes to smaller inner-city roads. This makes sense since all heuristics prioritize edges with high ply, which are more likely to lie on main roads.

▸ 5.4 Conclusion

We have introduced a new, data-driven approach to road network generalization. In particular, we introduce route-preserving road network generalization, when a road network and a collection of routes on the network is given. While the general problem—maximizing the number or length of routes represented—is NP-hard, there are good heuristics that preserve many routes, and also support continuous zooming. Our study of this approach is pure, in the sense that optimizing the number or length of routes fully covered is the only optimization criterion. It is surprising that we automatically get just one or very few connected components, and relatively few leaves. Although it would have been possible to define a heuristic enforcing only one connected component, by requiring every route picked after the first to have an

overlap of at least one vertex with the already picked subgraph, it seems that this is not needed.

It would be interesting to test our methods on other data sets, for instance, from different countries. Also, at the moment we do not know how large a collection of routes needs to be to let the approach perform well. We also do not know how much our results are influenced by the anonymization of the route data.

More generally, it would be very interesting to start a large-scale comparative study on the different road network generalization methods, as there are many. The methods can be compared on all relevant aspects, including quantitative ones, support of continuous zooming, and ideally even an expert study on the cartographic quality of the results for mapping purposes.

For further theoretical study of the problem, the next step is to look for approximation algorithms for the NP-hard problem cases, or proofs that these cannot exist. Variant problem formulations are also a possible topic of study. Our current problem formulation focuses on preserving as many routes as the budget allows and by varying the budget we can reduce the overall detail of the road network. This formulation is good for ensuring no important routes are lost, but there is no guarantee that the resulting road network is not visually cluttered on a local scale. Possible variant problem formulations to be studied could include some sort of threshold on local density of the selected subgraph.

Chapter 6

Conclusion

In this thesis, we have studied a variety of different problems involving trajectories.

In Chapter 2, we covered the problem of outliers in trajectories, and proposed new methods of removing outliers, even when minimal context for the trajectories is available. Experimental results show the proposed methods outperform the benchmark algorithms they were compared to.

In Chapter 3, we covered trajectory simplification, and showed for many variants of the problem whether they are NP-hard or we can prove they can be solved in polynomial time.

In Chapter 4, we looked at representative trajectories, and made the first implementation of the central trajectories algorithm, experimentally showing it has significantly better performance in practice than in theory.

In Chapter 5, we looked at an application task: generalizing a road network based on trajectory data. We formulated the route-preserving road network generalization problem and analyzed its complexity. We then introduced heuristical approaches for finding a data-driven generalization and tested them out experimentally.

While the different chapters each have their open problems remaining, leading to opportunities for future research, included below are some more general thoughts on where trajectory research could go in the future.

Integral approaches to preprocessing A common thread between the research in the preceding chapters is a focus on trajectory preprocessing. Two of the chapters directly involve preprocessing tasks and two of the chapters involve an intensive preprocessing process to transform trajectory databases into a set of trajectory clusters

and a set of paths on a road network. Preprocessing trajectories is of vital importance in order to be able to get good results with subsequent trajectory application tasks. The experiments for this thesis involved working with real trajectory data. Working with this data exposed many data quality issues in raw trajectory data sets: Missing data, identifiers being reused, extremely sparse trajectories, extremely short trajectories, and more. Besides the number of different errors that need fixing in a data set, there is also the question of the order in which to apply the preprocessing steps. Consider the following scenario: We are given a sparse data set of trajectories that involve cars driving between different towns. In their trips, the cars first drive on town roads with a low maximum speed, before turning onto a highway to travel to their destination town. The highway has a much higher maximum speed than the town roads, so if we want to use our outlier detection methods from Chapter 2, it will be difficult to choose a good value for the maximum speed parameter. If we set a high value we will miss outliers among the probes measured on town roads, but if we set a low value we will discard all probes measured on the highway. Ideally we would want to segment the trajectory so that we can process the parts of the trajectories captured on the highway separately when detecting outliers, with a higher maximum speed. However, any segmentation algorithm we want to apply will have to deal with the fact that the trajectory still has outliers in it, possibly influencing the result. Both of the algorithms depend on each others output in order to be able to run optimally. There are multiple of such chicken-and-egg scenarios arising when dealing with preprocessing steps. For these reasons, a future topic for theoretical study is in finding a more integral approach to trajectory preprocessing, where multiple tasks are handled either simultaneously, or sequenced in such a way that the algorithms chosen for the tasks work well together.

Alternative methods for tracking movement and indoor trajectories Besides GPS as a means of tracking movement, recently other methods have become more common, such as RFID- or Wifi-tracking. These methods can be particularly helpful for tracking movement of persons moving around indoors. After all, inside a building there are no clear lines of sight to GPS satellites. These indoor trajectories have an interesting context to their movement compared to well-studied trajectory contexts. Hurricanes, and animals with GPS trackers, can generally move freely in 2-dimensional space. Cars and outdoor pedestrians are usually bound to a road network. Pedestrians moving indoors are not bound by a road network, but their movement is highly constrained by the architecture of whatever building they are in due to where the walls and doors are placed. This type of context could prompt new problem formulations for well-studied trajectory tasks. When doing outlier

detection, a probe might be deemed an outlier even though it is spatially close to the rest of the trajectory, if it is separated by a wall. Likewise, when clustering trajectories, we may require all trajectories to traverse the same sequence of rooms in order to be part of the same cluster. Existing research on analysis of indoor trajectories includes Jin et al. [79], who search for regions of interest, Prentow et al. [118], who study indoor road network construction, and Jensen et al. [74], who look at trajectory indexing. Still, for many trajectory tasks there is no research yet for the indoor context. Algorithms specifically focused on handling trajectories captured indoors can therefore be an interesting topic of future study, both for when the exact layout of the building is known and for when it is unknown.

Dealing with anonymized data Nearly every recent paper on trajectory algorithms opens with the observation that the amount of available trajectory data has greatly increased over the past few years. This is true, and we can expect the amount of data released to keep increasing. However, it is possible that the quality of the data that will be made available for research will decrease over the coming years and the types of released data will change. The reason for this is another development that has taken place over the past years, namely an increasing concern in protecting privacy. As techniques for inferring private information from trajectory data improve, it will become harder to publish trajectory data generated by tracking people without running into privacy issues. Techniques for anonymization of the data will improve as well, but since there is always a trade-off between privacy and usability of the data [129] we can expect the usability of published data sets to decrease in the long run. Digital technology can change rapidly and recently, legislative bodies such as the European Union also see privacy as a topic of increasing concern, so these adverse effects on data set usability may arrive sooner than expected. For researchers, this may imply an increasing dependence on proprietary data sets, which is not good for reproducability. It also means that future research should be done into algorithms that take anonymized trajectories as input. Although anonymized data has reduced usability for application tasks, the anonymization method might allow us to make strong assumptions on the input, which can help with algorithm design. For example, only small subsequences of trajectories could be published, which would give increased importance on algorithms that assume a maximum trajectory length. Trajectories might be published k-anonymously [111], which for our method for road network generalization presented in Chapter 5 would mean the ply for any edge would be at least k, or zero. For trajectory clustering, we might consider each set of k-anonymous trajectories its own cluster, or we may still wish to cluster different sets together. We could compute the representative trajectory of such a

cluster with the same algorithm as used in Chapter 4, but we could also consider some weight-based method, where each set of identical trajectories is treated as a single trajectory with a weight proportional to the number of trajectories in the set. Only publishing representative trajectories and the number of trajectories they are based on could also be seen as another layer of anonymization.

Depending on which anonymization techniques become the most widely used different approaches will be needed, so collaboration between academia and industry will be of great importance to set the research agenda.

Integration of heterogeneous data sources This thesis mainly focuses on trajectory data. However, this is not the only type of data available for studying movement. For example, to record car movements, many roads contain loop detectors which count the number of vehicles using the road. Semi-autonomous cars collect LIDAR data for navigation that can also be used for tracking their movement. Contextual data, such as the underlying road network, can also deepen our understanding of the movement data. Combining two heterogeneous data sources is challenging, since data of different forms require different methods for handling them, and the data sources may have their own errors and uncertainties to account for. The result of a successful combination, however, can be very useful. For example, the outlier detection method presented in Chapter 2 could benefit from knowledge of internal measurements a car being tracked. If a measured speed is available for each probe, the physics model with both a speed and acceleration bound becomes concatenable, which can greatly speed up detecting the outliers. As another example, map matching combines GPS data with a road network and has turned out to be a very useful way of removing errors from the trajectories, among other uses. The combination of GPS and road network data has been the subject of much study already, e.g. in Chapter 5 of this thesis. There has been some recent research into combining LIDAR and GPS data for use in navigating [60, 41] as well as a study into combining GPS and loop detector data [45]. However, here is still more work that can be done on integrating GPS and non-GPS data.

Existential theory of the reals In Chapters 3 and 5 we give NP-hardness proofs for many different theoretical problems. For not all of these problems, however, are we able to also say whether the problems are in NP. While solutions to instances of these problems that have polynomial complexity can easily be verified, there can possibly be solutions requiring exponential bit complexity. Many problems in computational geometry run in to this issue, such as packing [8], or the art gallery problem [7]. For problems where it has been shown this exponential bit phenomenon

occurs, there exists the complexity class $\exists\mathbb{R}$. Problems which are $\exists\mathbb{R}$-hard are at least as difficult as the problem known as the *existential theory of the reals* [121]. Either all $\exists\mathbb{R}$-complete problems fall within NP, or none of them do. There has been an increasing amount of study into this complexity class, as determining if a single problem with the exponential bit phenomenon is in NP is asking if NP= $\exists\mathbb{R}$. So studying these problems together focuses on the core of the issue [121]. Recently, Erickson et al. [53] have used smooth analysis to show that for many $\exists\mathbb{R}$-complete problems, the exponential bit phenomenon only occurs for degenerate input. It would be interesting future work to see if the problems in this thesis where we were not able to show NP-membership are $\exists\mathbb{R}$-hard and to see if the result from [53] applies. More generally, for future theoretical research into trajectory problems it is good to not only consider if a problem is NP-hard but to consider $\exists\mathbb{R}$-hardness as well.

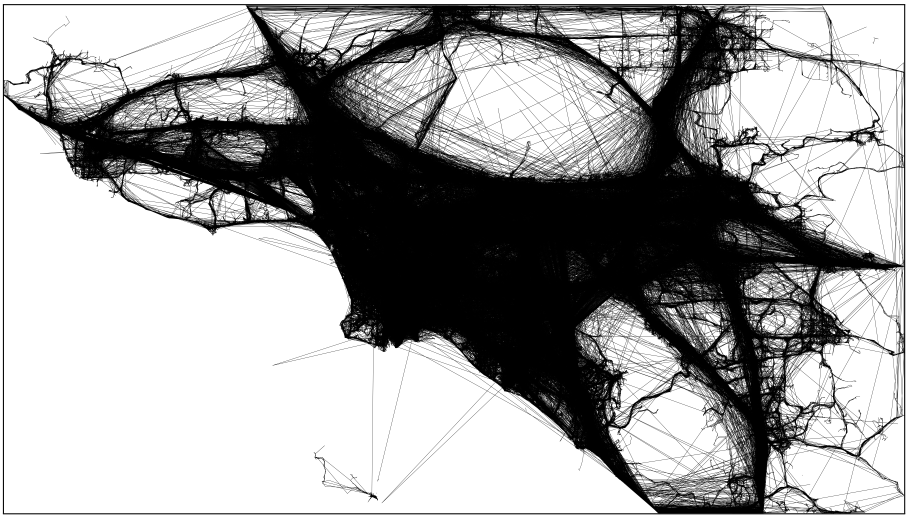
Appendix A

Data Sets

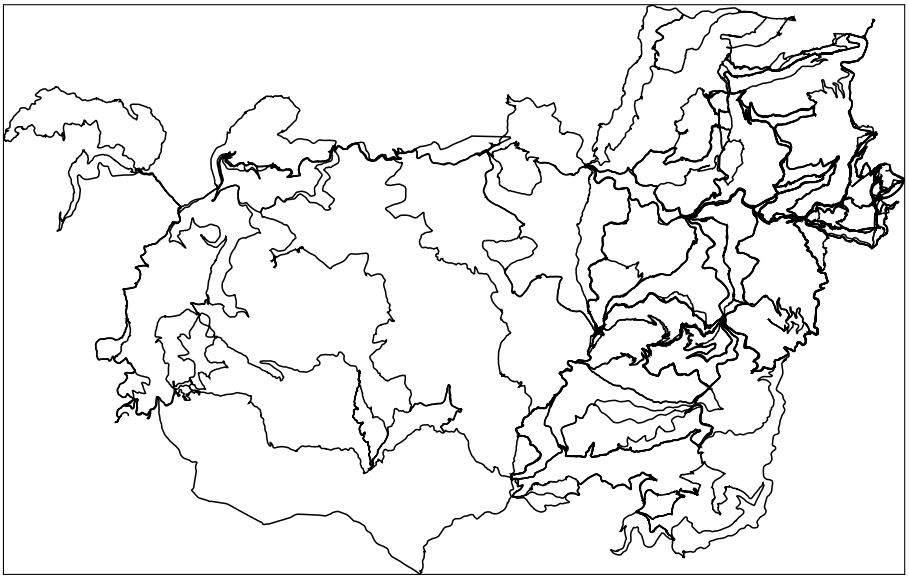
Chapters 2, 4, and 5 contain experiments performed on real trajectory data sets. Below we will describe the data sets used in the different chapters. Appendices B and C go over the preprocessing steps that were applied to these data sets to prepare them for use in chapters 4 and 5 respectively.

South Holland This is a proprietary data set provided by HERE Technologies. It consists of car and truck trips taken in the Dutch province of South Holland on a single day in January 2019. This data set is very large, so for our experiments we did not use the full data set. Instead, we have used different subsets of the data set. For Chapters 2 and 4 we have taken a random selection of 5000 trajectories from the data set. We will refer to this subset as HR (for the Hague and Rotterdam, the two biggest cities in South Holland). For Chapter 5 we have created two subsets of the data based on selecting all of the trajectories falling within a bounding box. One subset, which contains trajectories in a bounding box roughly corresponding to the region known as the "Groene Hart" ("*Green Heart*"), we will refer to as GH. The other subset contains the trajectories that lie north of GH. We will refer to this subset as NSH (North- South Holland). For a visual sample of HR and the clusters that are extracted from it for Chapter 4, see Figures A.2(a) and A.2(b). See Figure A.3 for visualizations of NSH and GH and the routes extracted for Chapter 5.

Los Angeles This is another proprietary data set provided by HERE Technologies. It contains car and truck trips taken in Los Angeles, California, on a single day in September 2018. We will refer to this data set as LA. See Figure A.1(a) for a

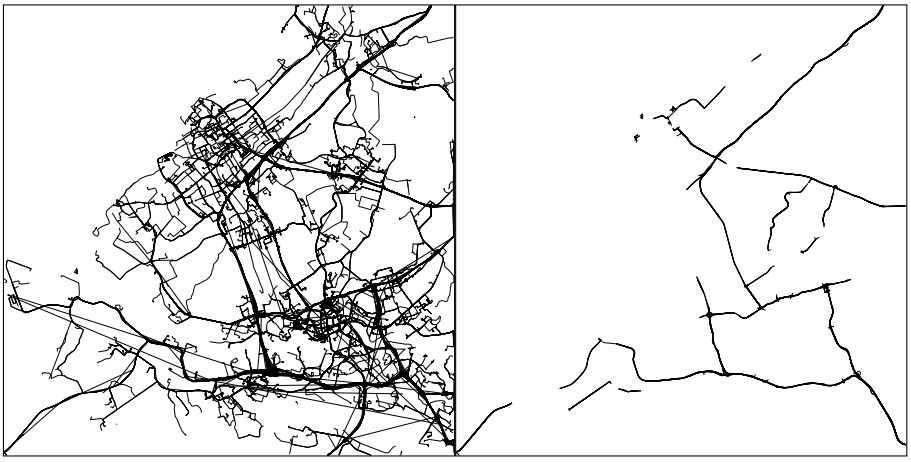


(a) LA



(b) MB

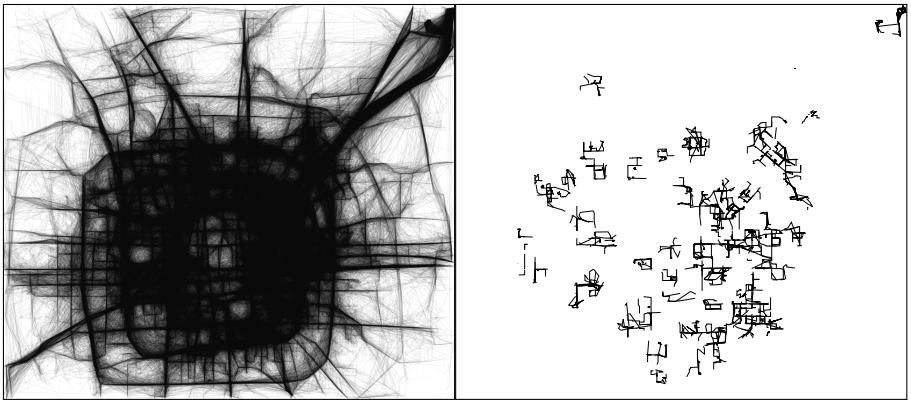
Figure A.1: Samples of the data sets used in Chapter 2.



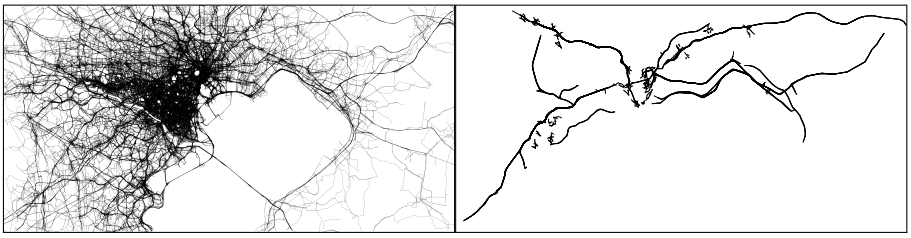
(a) HR (b) HR Clusters

(c) GL (d) GL Clusters

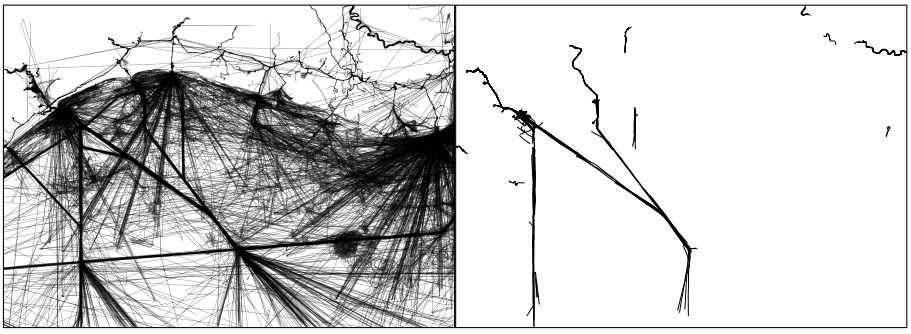
Figure A.2: Samples of the data sets used in Chapter 4. Figure continued on next page.



(e) TD (f) TD Clusters



(g) PF (h) PF Clusters



(i) MC (j) MC Clusters

Figure A.2: Samples of the data sets used in Chapter 4.

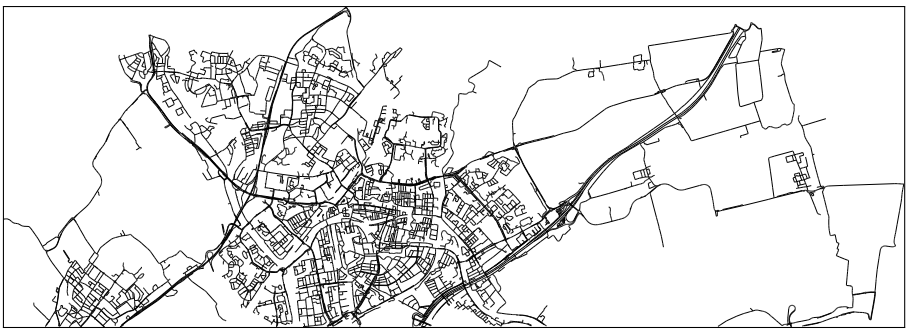
(a) GH

Figure A.3: The data sets used in Chapter 5. Figure continued on next page.

(b) GH routes

(c) NSH

Figure A.3: The data sets used in Chapter 5. Figure continued on next page.



(d) NSH routes

Figure A.3: The data sets used in Chapter 5.

visualization.

Mountainbike trips This is a private data set of one cyclist's mountain bike trips in several European countries, collected between 2012 and 2019. We will refer to this data set as MB. See Figure A.1(b) for a sample visualization.

T-drive This is a publicly available data set [135, 134] of taxi trajectories, collected in Beijing in one week, from February 2 to February 8 in 2008. We will refer to this data set as TD. For visual samples of the data set and the clusters that are extracted from it for Chapter 4, see Figures A.2(e) and A.2(f).

GeoLife This is a publicly available data set [142, 140, 141], created by tracking 182 people over a longer period between April 2007 to August 2012. The trajectories record many different outdoor activities and also different types of transport. We will refer to this data set as GL. For visual samples of the data set and the clusters that are extracted from it for Chapter 4, see Figures A.2(c) and A.2(d).

OpenPFlow OpenPFlow [82] is a method for generating a publicly publishable data set by simulating agents based on input data. This avoids the privacy problems and cost associated with gathering and publising real trajectory data. The authors of the paper introducing it have used their method to generate a data set based on data on the Tokyo metropolitan area, which we have also used. We refer to this data of the Tokyo area as the OpenPFlow data set. We will refer to this data set as

Table A.1: Statistics on the raw data sets.

Data set	# trajectories	Avg. # probes	Avg. length	Avg. duration
HR	5,000	425	20,810m	1,765s
GH	41,961	52	7,899m	495s
NSH	17,285	72	6,774m	722s
LA	124,234	194	56,907m	24,172s
MB	1,214	3,377	37,267m	7,623s
TD	10,336	1,708	7,316km	507,158s
GL	17,784	1,398	328,768m	9,492s
PF	83,333	1,195	21,391m	71,704s
MC	6,398	11,607	1,106km	1,644,180s

PF. For visual samples of the data set and the clusters that are extracted from it for Chapter 4, see Figures A.2(g) and A.2(h).

MarineCadastre `MarineCadastre.gov` is a site by the American government that provides, among other things, large amounts of GPS data obtained by tracking ships near the US. We have used their data from April 2007, located in the 15th UTM zone: This is the area of the earth between -96 and -90 degrees longitude. We will refer to this data as the MarineCadastre data set. We will refer to this data set as MC. For visual samples of the data set and the clusters that are extracted from it for Chapter 4, see Figures A.2(i) and A.2(j).

Appendix B

Preprocessing for Chapter 4

For this chapter, we use the data sets HR, TD, GL, PF and MC. For our experiments we want to apply the Central Trajectories algorithm, which requires trajectory clusters as input. So we will have to extract the clusters from these data sets. This also requires some other preprocessing.

B.1 Preprocessing steps before clustering

The full data sets are much larger than is practical for our experiment, so first a random subset was taken from all of the data sets to reduce the amount of data to an amount that would not take too long to process. Since the length of individual trajectories is a large factor in the runtime of the clustering algorithm, for data sets with very long trajectories the trajectories were split up into sections with a maximum number of probes n_{max}, where the sections are treated as individual trajectories. Then, the trajectories are filtered so that only trajectories with a bounding box of at least a given area A_{min} are included. We do this to filter out trajectories that mostly consist of stay points with little actual movement of the entity, as those trajectories can't be properly clustered with the used algorithm. Furthermore, the T-drive data set was split up so that each trajectory covers the movement of the entity over a single day. Because the clustering algorithm does not scale well we do not cluster an entire data set at once, instead, we pick a point in the bounding box of the data set and then find the k trajectories that are closest to that point, and cluster that set. We keep picking new points and finding new blocks of trajectories to cluster until we have found enough clusters. We aimed to get around 100 clusters for each data

set. Although we risk splitting up some clusters with this approach, it saves a lot of time since the clustering algorithm no longer has to consider pairs of trajectories that were not spatially close in the first place.

B.2 Clustering algorithm

To cluster the data, we make use of the Subtrajectory Clustering algorithm by Buchin et al. [32]. The method finds a subtrajectory cluster occuring within one long trajectory. This algorithm requires two parameters: a maximum bound on the Fréchet distance in a cluster δ and a minimum cluster size $|C|_{min}$. It works by constructing a free space diagram between the trajectory and itself for distance δ. Then, an interval on the x-axis of the free space diagram is maintained. The interval starts at the leftmost point of the free space diagram with a width of 0. At any point we only consider the section of the free space diagram where the x-coordinate lies in the interval, called the interval's *vertical slab*; As long as there are at least $|C|_{min}$ paths connecting the left and right ends of the vertical slab that are disjoint when projected on the y-axis, the interval is grown. If there are no longer enough paths, the interval is shrunk down to minimum size and moved to the right. After the entire free space diagram is traversed the longest interval that was found is returned. We use the implementation in MoveTK [2] which uses the discrete Fréchet distance. To use the algorithm for our purposes we have made some adaptations, which we will now explain.

(i) The unadapted algorithm works for finding the largest cluster of subtrajectories from a single trajectory. We want to find clusters in a data set of multiple trajectories, so we extend the algorithm in a manner proposed by Buchin et al. themselves for the case with multiple trajectories. This means we append all of the trajectories together into one very large trajectory which we will call the *supertrajectory*. To make sure the algorithm does not accidentally return a cluster where one of the subtrajectories contains pieces of multiple input trajectories, between the endpoints of trajectories we are appending we first insert one dummy probe that is far removed from any of the trajectory probes.

(ii) The unadapted algorithm does not find all possible clusters, but it only finds the longest cluster. Additionally, creating a free space diagram between the supertrajectory and itself requires an impractically large amount of memory. To deal with these issues, what we do instead is create a free space diagram between each individual input trajectory and the supertrajectory. For each of these diagrams we will run the algorithm, and so for each input trajectory

we will find the largest cluster that the trajectory is involved in. Then we filter this obtained set of clusters: we first sort the clusters by length and then iteratively add the longest cluster to our final set of clusters. If we add a cluster, we mark all of the probes as taken. Any cluster that comes after it will have any subtrajectories that include already taken probes filtered out, and only if the cluster still meets the threshold for minimum size and length after this filtering it will be added to the final list of clusters.

(iii) The unadapted algorithm creates a free space diagram of the supertrajectory with itself, it then finds the largest interval on the x-axis of the free space diagram where the vertical slab of the free space diagram on that interval contains at least the minimum cluster size $|C|_{min}$ distinct routes in the free space between the left and right ends of the slab. I.e., it optimizes clusters for the length of the longest subtrajectory in the cluster. The other routes in the free space are made as short as possible, since the algorithm only cares about satisfying a minimum number of routes, and a shorter route gives a greater probability of finding another route since there is more of the diagram left to cover. For data sets that require a large distance threshold, or data sets with trajectories that contain stay points, this has an unfortunate effect of producing clusters that have one long subtrajectory (based on the trajectory whose diagram is being handled) and otherwise only contain subtrajectories of just a single probe. To combat this, we have adapted the algorithm so any routes we find in the free space are made as long as possible instead, by appending more probes as long as the distance threshold is still respected. Then, any routes that are shorter than a minimum probe threshold $|T|_{min}$ are discarded and only clusters that contain a minimum number of subtrajectories that are long enough are considered.

After all of these adaptations the algorithm no longer necessarily returns any optimal clusters in the data set, but we do know that all of the returned clusters are actually present in the data. Since our research in Chapter 4 focuses on central trajectories, and not the clustering itself, we consider this acceptable as long as we are able to extract enough usable clusters from the data. For each data set we set the minimum cluster size $|C|_{min}$ to 3. The other preprocessing parameters are given in Table B.1.

▸ B.3 Additional preprocessing

The central trajectories algorithm requires input trajectories to have the same number of probes with the same timestamps. The clustering algorithm, however, doesn't

Table B.1: Parameters for the preprocessing applied to the data sets. The columns give the maximum trajectory size after splitting long trajectories n_{max}, cluster algorithm block size k, maximum discrete Fréchet distance δ (in meters), the minimum bounding box area for trajectories A_{min} ($x10,000,000$, in m^2), and the minimum number of probes for any subtrajectory in a cluster $|T|_{min}$.

Raw data set	n_{max}	k	δ	A_{min}	$\|T\|_{min}$
HR	-	100	300	0	20
TD	-	100	1000	280	10
GL	2000	100	1000	10	20
PF	2000	100	1000	23	20
MC	2000	50	9000	1850	50

take time into account at all, so the subtrajectories in the found clusters are not necessarily close in time, let alone sampled equally. To prepare the obtained clusters for the central trajectories algorithm, we first shift the timestamps of the subtrajectories in the cluster so that they start at the same time. Then, we perform an *elementary resample*, where we first truncate the trajectories to only include probes in the time interval of the subtrajectory with the shortest time interval, and then use linear interpolation to add a probe for any timestamp that is present in another subtrajectory in the cluster. After this step, the clusters are ready to be used by the central trajectories algorithm.

▸ B.4 Final data sets

After all of this preprocessing we obtained five data sets, each containing 94-160 clusters of between 3 and 31 trajectories with 9-1226 probes. Some statistics about these data sets are given in Table B.2.

Table B.2: The trajectory type, trajectory count, average number of probes per trajectory, average trajectory length in meters, number of clusters and average number of trajectories per cluster for each preprocessed data set used in Chapter 4. The average number of probes is based on the data after colinear probes have been removed.

Final data set	**HR**	**TD**	**GL**	**PF**	**MC**
Trajectory type	car	taxi	mixed	pedestrian	ship
Traj. count	719	580	688	299	390
Avg. #probes	41.98	5.29	18.02	17.25	60.84
Avg. length	3195m	729m	2851m	12686m	22190m
#Clusters	160	108	136	81	94
Avg. clus. size	4.49	5.37	5.06	3.69	4.15

Appendix C

Preprocessing for Chapter 5

For the experiments in this chapter, we need two types of data: a road network, and a set of routes on the network.

As our road network, we use the freely-available OpenStreetMap (OSM) [115]. To prepare the map for our experiments, we clipped the map to rectangular sections around our areas of interest where we have trajectory data, and further restricted the road network to those sections that are actually used by at least one of the routes. Finally, we retain only the largest connected component of the remaining network in each area.

We use GH and NSH as a source for routes.

Since routes are not provided as paths in the graph but raw GPS coordinates, we first run them through a map-matching algorithm. We used the GraphHopper system [64], which is based on the hidden-Markov method by Newson and Krum [112].

After mapmatching, the trajectories still needed to be transformed into (combinatorial) paths in the graph. GraphHopper outputs GPS points that mostly lie on graph vertices. To transform the trajectories into paths we matched each GPS point to the closest graph vertex, and then took the shortest path from vertex to vertex. Almost all points could be closely matched to a vertex this way, but the first and last points of trajectories were not always close to vertices as the trajectory would end before the tracked vehicle reached a vertex location. To avoid the risk of getting bad matches, we opted to remove the first and last points of trajectories if they lay more than 10 meters from the nearest vertex.

After this step, we removed duplicate edges from the trajectories, so that each trajectory covers each edge at most once, as we do not want one trajectory covering

Table C.1: Specifications for the data sets used in Chapter 5.

Data set	GH	NSH
Latitude range	51.9696–52.1389	52.1389–52.2038
Longitude range	4.42141–4.7177	4.37087–4.66767
# Vertices	36,898	25,716
# Leaf vertices	1054	1064
# Edges	39,071	27,628
Total edge length	1.284×10^6 m	776,636 m
# Routes	27,560	13,227
Avg. route length	8401 m	8712 m
Std. dev. route length	5340 m	3858 m

an edge many times being able to inflate the importance of an edge. We removed any trajectory that is less than 1000 meters in length, as we only want to count significant trips.

The specifications for the final data sets can be found in Table C.1.

Acknowledgements

I have been working on this PhD thesis for the past four years. During this time, I have depended on the help of many people and have experienced the kindness of many more. Here, I would like to express my gratitude to them. I have not spent as much time with everyone as I would have wished, especially due to circumstances forcing me to work from home during the last year and eight months; but the time we did share I will cherish for a long time.

First of all my gratitude goes to my supervisors: Marc van Kreveld, Irina Kostityna, and Maarten Löffler. Thank you for the great opportunity that this PhD journey has been. Thanks as well for the kind words and thoughtful approach when supervising, and for all of the help with and feedback on my work.

My thanks go out to the members of my reading committee, Kevin Buchin, Anne Driemel, Peter van Oosterom, Arno Siebes, and Remco Veltkamp, for taking the time to read this thesis and give feedback.

I want to thank all of my colleagues from the Geometric Computing group at Utrecht University. In particular I want to thank Ivor van der Hoog, Alex Lewis, Jérôme Urhausen, and Jordi Vermeulen for the great games of 7 Wonders, and the companionship we've had since starting our PhDs at around the same time. I also want to thank Wouter van Toll, Arne Hillebrand and Tillman Miltzow who I was officemates with -when that was still a thing- for their pleasant company and advice.

My thanks also go out to all of the people I have done research with. Of the people not named already, this is Bram Custers, Bettina Speckmann, Wouter Meulemans, Frank Staals, Carola Wenk, Majid Mirzanezhad, Sampson Wong, Joachim Gudmundsson, Lionov Wiratma, André van Renssen, and Roald Melssen. I had a great time working with you all and I am proud of what we were able to achieve together.

During my travels to workshops and conferences I have met many great people whom have shown me kindness and have given me good conversations. There are

too many people to name individually but my thanks go out to you all.

I want to thank Aniket Mitra and Onur Derin for their help with the implementations of some of the algorithms.

Besides all of the people in academia named above I have of course also spent time with and received support from friends and family. Thank you for your support and your belief in my ability to complete this PhD journey. In particular I want to thank my parents, for always being there for me and supporting me in all of my pursuits. Last but definitely not least I give thanks to my girlfriend, Cath. Thank you for all of your love and support, your belief in me, and for putting up with me spending half of my PhD working from our living room. I love you.

Curriculum Vitae

Mees van de Kerkhof was born on 29 July 1993 in Beuningen, in the province of Gelderland in the Netherlands. He finished his secondary education in 2011 at the Dominicus College in Nijmegen. In 2014 he received his Bachelor's degree in Computer Science from Utrecht University. After pausing his studies for one year to serve on the board of his student society, he followed with a Master in Computer Science, which he finished in December of 2017. His Master's thesis was on the automatic generation of curved nonograms, a novel type of pen-and-paper puzzle. It was supervised by prof. dr. Marc van Kreveld, dr. Maarten Löffler, and dr. Amir Vaxman. As of December 2017 he has been working as a PhD student in the Geometric Computing group, as part of the Algorithms division at Utrecht University. The results of his PhD are presented in this thesis.

Other publications In addition to the publications incorporated into this thesis, Mees has published the following formally reviewed articles:

[83] V. Keikha, M. van de Kerkhof, M. van Kreveld, I. Kostitsyna, M. Löffler, F. Staals, J. Urhausen, J. L. Vermeulen, and L. Wiratma. Convex partial transversals of planar regions. In *29th International Symposium on Algorithms and Computation, ISAAC 2018*, pages 52:1–52:12. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2018.

[84] M. van de Kerkhof, T. de Jong, R. Parment, M. Löffler, A. Vaxman, and M. van Kreveld. Design and automated generation of Japanese picture puzzles. *Computer Graphics Forum*, 38(2):343–353, 2019.

[65] J. Gudmundsson, M. van de Kerkhof, A. van Renssen, F. Staals, L. Wiratma, and S. Wong. Covering a set of line segments with a few squares. In *International Conference on Algorithms and Complexity*, pages 286–299. Springer, 2021.

[70] I. van der Hoog, M. van de Kerkhof, M. van Kreveld, M. Löffler, F. Staals, J. Urhausen, and J. L. Vermeulen. Mapping multiple regions to the grid with bounded Hausdorff distance. In *Workshop on Algorithms and Data Structures*, pages 627–640. Springer, 2021.

Samenvatting

Sinds het begin van deze eeuw is er sprake van een explosieve groei in het gebruik van GPS. Het wordt onder andere gebruikt in navigatiesystemen van auto's, en ook door smartphones om de locatie van de telefoon vast te stellen. Als een GPS-tracker gebruikt wordt om langere tijd de locatie van hetzelfde mens, dier, of object te volgen is het resultaat een zogeheten *trajectory*. Dit is een verzameling tijd-en-positiemetingen. Als we de gemeten posities met rechte lijnstukken met elkaar verbinden krijgen we een benadering van de route die de tracker heeft afgelegd. Door het grootschalige gebruik van GPS is er veel trajectory-data beschikbaar. Analyse van trajectory's kan helpen om meer te begrijpen van de bewegingen van hetgeen dat gevolgd wordt. Trajectory-databases zijn in de regel echter zo groot dat handmatige analyse niet werkbaar is. Het is beter om slimme algoritmes te ontwikkelen die analyse kunnen automatiseren. Ook zijn er algoritmes voor de voorbehandeling van de data, zoals het corrigeren van fouten of het aanbrengen van structuur in de database, enz. Dit proefschrift gaat over vier verschillende vraagstukken die te maken hebben met het verwerken van trajectory-data. Het bevat een combinatie van theoretisch onderzoek naar algoritmes die de vraagstukken moeten oplossen, en praktijkonderzoek waarbij algoritmes worden getest op echte trajectory-databases. De vraagstukken waar we naar kijken zijn de volgende:

- In Hoofdstuk 2 kijken we naar het opsporen van zogeheten *outliers*. Dit zijn metingen in de trajectory waarvan reden is aan te nemen dat ze het resultaat zijn van een meetfout. Dit kan bijvoorbeeld gebeuren als de GPS-tracker in een omgeving is met veel hoge gebouwen die de signalen van GPS-satellieten blokkeren. We introduceren efficiënte algoritmes voor dit probleem die outliers opsporen door te kijken naar de fysieke eigenschappen van het gevolgde object. Als een trajectory van een gevolgde auto bijvoorbeeld twee opeenvolgende metingen heeft die zo ver uit elkaar liggen dat de auto sneller dan zijn maximumsnelheid zou hebben moeten rijden om de metingen te laten

kloppen, dan is één van de twee metingen waarschijnlijk een outlier. We introduceren drie verschillende algoritmes voor verschillende fysieke limieten die een gevolgd object kan hebben. Vervolgens vergelijken we deze algoritmes experimenteel met simpelere alternatieven en laten zien dat onze aanpak beter werkt.

- In Hoofdstuk 3 kijken we naar het versimpelen van trajectory's, en algemener gezien het versimpelen van iedere kromme die bestaat uit een verzameling punten die met lijnstukken verbonden zijn. Met versimpelen wordt bedoeld dat we een andere trajectory/kromme vinden die uit minder metingen/punten bestaat maar er toch zo goed als hetzelfde uitziet. Zo hebben we minder geheugen nodig om de data op te slaan, en kunnen ze ook sneller algoritmisch verwerkt worden. Er zijn zeer veel verschillende versies van het versimpelingsvraagstuk, afhankelijk van hoe je meet hoeveel twee verschillende krommen op elkaar lijken, en van welke andere eisen je aan de nieuwe kromme stelt. We behandelen meerdere van deze varianten en geven van iedere variant een efficiënt algoritme, of een bewijs dat er geen efficiënt algoritme bestaat dat voor die variant de beste oplossing kan vinden.

- In Hoofdstuk 4 kijken we naar het vinden van een representatief trajectory voor een cluster trajectory's. Een cluster is een groep trajectory's die meer gemeen hebben met elkaar dan met de andere trajectory's in de database. Een cluster zou bijvoorbeeld kunnen bestaan uit een groep trajectory's van auto's die allemaal ongeveer dezelfde route hebben afgelegd. Om in een oogopslag te kunnen zien hoe de trajectory's in het cluster ongeveer lopen willen we graag een enkel trajectory vinden dat representatief is voor het cluster. Dit kan een trajectory uit het cluster zijn, maar we kunnen ook algoritmisch een nieuw trajectory construeren dat zo representatief mogelijk is. In dit hoofdstuk kijken we naar een bestaand algoritme voor het maken van zo'n representatief trajectory. Door experimenten onderzoeken we hoe de resultaten van het algoritme in de praktijk zijn ten opzichte van de theorie.

- In Hoofdstuk 5 kijken we naar het generaliseren van een wegennetwerk met behulp van trajectory's. Als we een wegennetwerk willen laten zien, willen we vaak niet dat iedere weg, hoe klein ook, zichtbaar is. Als het plaatje ver genoeg uitgezoomd is wordt dat immers erg druk en zijn de details overweldigend. Daarom willen we, naarmate de schaal van de visualisatie van het wegennetwerk groter wordt, steeds meer wegen weglaten. Dit noemen we het generaliseren van het netwerk. Wel willen we uiteraard dat de belangrijkste wegen nog steeds zichtbaar zijn, ook op grote schaal. Door gebruik te

maken van trajectory-data kunnen we automatisch achterhalen welke wegen belangrijk zijn. Deze wegen zullen immers door meer trajectory's gevolgd zijn. We introduceren hiervoor in dit hoofdstuk een specifieke probleemstelling: Gegeven het percentage van de totale lengte van het wegennetwerk dat behouden mag blijven, kies welke wegen blijven zo, dat het aantal trajectory's dat volledig op behouden wegen ligt maximaal is. We doen eerst theoretisch onderzoek naar deze probleemstelling en geven een bewijs dat deze formulering van het probleem helaas niet optimaal opgelost kan worden op een efficiënte manier. In experimenteel onderzoek bekijken we oplossingen die werken met een heuristiek, waardoor ze weliswaar niet optimaal zijn maar wel efficiënt berekend kunnen worden. We laten zien dat we heuristisch een redelijke oplossing kunnen geven, en dat onze probleemstelling tot minder onvolkomenheden leidt dan een andere mogelijke probleemstelling.

Samenvattend, dit proefschrift beschouwt een verscheidenheid aan taken op trajectory-data die met algoritmes opgelost kunnen worden, en hoe efficiënt die oplossingen zijn. Er wordt daarbij gekeken naar de theoretische kant, maar door middel van experimenten op echte data ook naar de toepassing van de algoritmes in de praktijk. Daarmee draagt dit proefschrift bij aan een beter begrip van trajectory-data en hoe deze gebruikt kan worden.

Bibliography

[1] https://www.gps.gov/systems/gps/performance/accuracy/.

[2] https://movetk.win.tue.nl/.

[3] https://www.cgal.org/.

[4] https://www.boost.org/.

[5] https://git.science.uu.nl/M.A.vandeKerkhof/centraltrajectoriesoutputdata.

[6] M. A. Abam, M. de Berg, P. Hachenberger, and A. Zarei. Streaming algorithms for line simplification. *Discrete & Computational Geometry*, 43(3):497–515, 2010.

[7] M. Abrahamsen, A. Adamaszek, and T. Miltzow. The art gallery problem is ∃ℝ-complete. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 65–73, 2018.

[8] M. Abrahamsen, T. Miltzow, and N. Seiferth. Framework for ∃ℝ-completeness of two-dimensional packing problems. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1014–1021. IEEE, 2020.

[9] P. K. Agarwal, S. Har-Peled, N. Mustafa, and Y. Wang. Near linear time approximation algorithm for curve simplification. *Algorithmica*, 42(3–4):203–219, 2005.

[10] M. Ahmed, S. Karagiorgou, D. Pfoser, and C. Wenk. A comparison and evaluation of map construction algorithms using vehicle tracking data. *GeoInformatica*, 19(3):601–632, 2015.

[11] M. Ahmed and C. Wenk. Constructing street networks from GPS trajectories. In *European Symposium on Algorithms*, pages 60–71. Springer, 2012.

[12] H. A. Akitaya, F. Brüning, E. Chambers, and A. Driemel. Covering a curve with subtrajectories. *arXiv preprint arXiv:2103.06040*, 2021.

[13] H. A. Akitaya, M. Buchin, L. Ryvkin, and J. Urhausen. The k-Fréchet distance: How to walk your dog while teleporting. In *30th International Symposium on Algorithms and Computation (ISAAC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2019.

[14] T. Akiyama, T. Nishizeki, and N. Saito. NP-completeness of the Hamiltonian cycle problem for bipartite graphs. *Journal of Information Processing*, 3:73–76, 1980.

[15] H. Alt, A. Efrat, G. Rote, and C. Wenk. Matching planar maps. *Journal of Algorithms*, 49(2):262–283, 2003.

[16] H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *International Journal of Computational Geometry & Applications*, 5(1-2):75–91, 1995.

[17] G. Andrienko, N. Andrienko, G. Fuchs, and J. M. C. Garcia. Clustering trajectories by relevant parts for air traffic analysis. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):34–44, Jan 2018.

[18] N. Andrienko, G. Andrienko, J. M. C. Garcia, and D. Scarlatti. Analysis of flight variability: a systematic approach. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):54–64, Jan 2019.

[19] B. Aronov, A. Driemel, M. van Kreveld, M. Löffler, and F. Staals. Segmentation of trajectories on nonmonotone criteria. *ACM Transactions on Algorithms (TALG)*, 12(2):1–28, 2015.

[20] A. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. *Algorithmica*, 1(1–4):49–63, 1985.

[21] G. Barequet, D. Z. Chen, O. Daescu, M. T. Goodrich, and J. Snoeyink. Efficiently approximating polygonal paths in three and higher dimensions. *Algorithmica*, 33(2):150–167, 2002.

[22] J. L. Bentley and J. B. Saxe. Decomposable searching problems I. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[23] S. A. Benz and R. Weibel. Road network selection for medium scales using an extended stroke-mesh combination algorithm. *Cartography and Geographic Information Science*, 41(4):323–339, 2014.

[24] S. Bereg, M. Jiang, W. Wang, B. Yang, and B. Zhu. Simplifying 3D polygonal chains under the discrete Fréchet distance. In *Proc. Latin American Symposium on Theoretical Informatics (LATIN)*, Lecture Notes in Computer Science (LNCS, volume 4957), pages 630–641, 2008.

[25] M. de Berg, M. van Kreveld, and S. Schirra. Topologically correct subdivision simplification using the bandwidth criterion. *Cartography and Geographic Information Systems*, 25(4):243–257, 1998.

[26] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval*, pages 39–48, 2000.

[27] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1978.

[28] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *Proc. 31st Conference on Very Large Data Bases (VLDB)*, pages 853–864, 2005.

[29] K. Bringmann and B. R. Chaudhury. Polyline simplification has cubic complexity. *Journal of Computational Geometry*, 11(2):94–130, 2021.

[30] K. Buchin, M. Buchin, D. Duran, B. T. Fasy, R. Jacobs, V. Sacristan, R. I. Silveira, F. Staals, and C. Wenk. Clustering trajectories for map construction. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–10, 2017.

[31] K. Buchin, M. Buchin, J. Gudmundsson, J. Hendriks, E. H. Sereshgi, V. Sacristán, R. I. Silveira, J. Sleijster, F. Staals, and C. Wenk. Improved map construction using subtrajectory clustering. In *Proceedings of the 4th ACM SIGSPATIAL Workshop on Location-Based Recommendations, Geosocial Networks, and Geoadvertising*, pages 1–4, 2020.

[32] K. Buchin, M. Buchin, J. Gudmundsson, M. Löffler, and J. Luo. Detecting commuting patterns by clustering subtrajectories. *International Journal of Computational Geometry & Applications*, 21(03):253–282, 2011.

[33] K. Buchin, M. Buchin, M. van Kreveld, M. Löffler, R. I. Silveira, C. Wenk, and L. Wiratma. Median trajectories. *Algorithmica*, 66(3):595–614, 2013.

[34] K. Buchin, S. Cabello, J. Gudmundsson, M. Löffler, J. Luo, G. Rote, R. I. Silveira, B. Speckmann, and T. Wolle. Finding the most relevant fragments in networks. *Journal of Graph Algorithms and Applications*, 14(2):307–336, 2010.

[35] K. Buchin, M. Löffler, A. Popov, and M. Roeloffzen. Fréchet distance between uncertain trajectories: Computing expected value and upper bound. In *36th European Workshop on Computational Geometry (EuroCG 2020)*, 2020.

[36] L. Buzer. Optimal simplification of polygonal chain for rendering. In *Proc. 23rd Annual ACM Symposium on Computational Geometry (SoCG)*, pages 168–174, 2007.

[37] J. Cardinal, S. Felsner, T. Miltzow, C. Tompkins, and B. Vogtenhuber. Intersection graphs of rays and grounded segments. *Journal of Graph Algorithms and Applications*, 22(2):273–295, 2018.

[38] E. W. Chambers, E. C. De Verdiere, J. Erickson, S. Lazard, F. Lazarus, and S. Thite. Homotopic Fréchet distance between curves or, walking your dog in the woods in polynomial time. *Computational Geometry*, 43(3):295–311, 2010.

[39] S. Chan and F. Chin. Approximation of polygonal curves with minimum number of line segments or minimum error. *International Journal of Computational Geometry & Applications*, 6(1):59–77, 1996.

[40] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12(1):54–68, 1994.

[41] D. Chen and G. X. Gao. Probabilistic graphical fusion of LiDAR, GPS, and 3D building maps for urban UAV navigation. *NAVIGATION*, 66(1):151–168, 2019.

[42] D. Z. Chen, O. Daescu, J. Hershberger, P. M. Kogge, N. Mi, and J. Snoeyink. Polygonal path simplification with angle constraints. *Computational Geometry*, 32(3):173–187, 2005.

[43] J. Chen, Y. Hu, Z. Li, R. Zhao, and L. Meng. Selective omission of road features based on mesh density for automatic map generalization. *International Journal of Geographical Information Science*, 23(8):1013–1032, 2009.

[44] M. Chimani, T. C. van Dijk, and J.-H. Haunert. How to eat a graph: Computing selection sequences for the continuous generalization of road networks. In *Proc. 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 243–252, 2014.

[45] B. Custers, W. Meulemans, B. Speckmann, and K. Verbeek. Route reconstruction from traffic flow via representative trajectories. *arXiv preprint arXiv:2012.05019*, 2020.

[46] B. Custers, M. van de Kerkhof, W. Meulemans, B. Speckmann, and F. Staals. Maximum physically consistent trajectories. *ACM Transactions on Spatial Algorithms and Systems*, 7(4):1–33, 2021.

[47] J. Czyzowicz, E. Kranakis, and J. Urrutia. A simple proof of the representation of bipartite planar graphs as the contact graphs of orthogonal straight line segments. *Information Processing Letters*, 66(3):125–126, 1998.

[48] P. Damaschke. The Hamiltonian circuit problem for circle graphs is NP-complete. *Information Processing Letters*, 32(1):1 – 2, 1989.

[49] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica*, 10(2):112–122, 1973.

[50] A. Driemel and S. Har-Peled. Jaywalking your dog: computing the Fréchet distance with shortcuts. *SIAM Journal on Computing*, 42:1830–1866, 2013.

[51] C. Duchêne, B. Baella, C. A. Brewer, D. Burghardt, B. P. Buttenfield, J. Gaffuri, D. Käuferle, F. Lecordix, E. Maugeais, R. Nijhuis, M. Pla, M. Post, N. Regnauld, L. V. Stanislawski, J. Stoter, K. Tóth, S. Urbanke, V. van Altenaand, and A. Wiedemann. Generalisation in practice within national mapping agencies. In D. Burghardt et al., editor, *Abstracting Geographic Information in a Data Rich World*, pages 329–391. Springer, 2014.

[52] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[53] J. Erickson, I. Van der Hoog, and T. Miltzow. Smoothing the gap between NP and ∃ℝ. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1022–1033. IEEE, 2020.

[54] S. P. Fekete, A. Hill, D. Krupke, T. Mayer, J. S. B. Mitchell, O. Parekh, and C. A. Phillips. Probing a set of trajectories to maximize captured information. In *18th International Symposium on Experimental Algorithms (SEA 2020)*, pages 5:1–5:14, 2020.

[55] M. Ferrante, C. Bongiorno, and N. Shoval. Similarity of GPS trajectories using dynamic time warping: An application to cruise tourism. In *Convegno della Società Italiana di Statistica*, pages 91–101. Springer, 2015.

[56] N. Ferreira, J. T. Klosowski, C. E. Scheidegger, and C. T. Silva. Vector field k-means: Clustering trajectories by fitting multiple vector fields. In *Computer Graphics Forum*, volume 32, pages 201–210. Wiley Online Library, 2013.

[57] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(1-4):153, 1987.

[58] Z. Fu, W. Hu, and T. Tan. Similarity based vehicle trajectory clustering and anomaly detection. In *IEEE International Conference on Image Processing 2005*, volume 2, pages 602–605. IEEE, 2005.

[59] S. J. Gaffney, A. W. Robertson, P. Smyth, S. J. Camargo, and M. Ghil. Probabilistic clustering of extratropical cyclones using regression mixture models. *Climate Dynamics*, 29(4):423–440, 2007.

[60] Y. Gao, S. Liu, M. M. Atia, and A. Noureldin. INS/GPS/LiDAR integrated navigation system for urban and indoor environments using hybrid scan matching algorithm. *Sensors*, 15(9):23286–23302, 2015.

[61] M. Gariel, A. N. Srivastava, and E. Feron. Trajectory clustering and an application to airspace monitoring. *IEEE Transactions on Intelligent Transportation Systems*, 12(4):1511–1524, 2011.

[62] Y. Ge, H. Xiong, Z.-h. Zhou, H. Ozdemir, J. Yu, and K. C. Lee. Top-eye: Top-k evolving trajectory outlier detection. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pages 1733–1736, 2010.

[63] M. Godau. A natural metric for curves – computing the distance for polygonal chains and approximation algorithms. In *Proc. 8th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, Lecture Notes in Computer Science book series (LNCS, volume 480), pages 127–136, 1991.

[64] GraphHopper. https://www.graphhopper.com.

[65] J. Gudmundsson, M. van de Kerkhof, A. van Renssen, F. Staals, L. Wiratma, and S. Wong. Covering a set of line segments with a few squares. In *International Conference on Algorithms and Complexity*, pages 286–299. Springer, 2021.

[66] J. Gudmundsson, M. van Kreveld, and F. Staals. Algorithms for hotspot computation on trajectory data. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 134–143, 2013.

[67] L. Guibas, J. Hershberger, J. Mitchell, and J. Snoeyink. Approximating polygons and subdivisions with minimum-link paths. *International Journal of Computational Geometry & Applications*, 3(4):383–415, 1993.

[68] M. Gupta, J. Gao, C. C. Aggarwal, and J. Han. Outlier detection for temporal data: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):2250–2267, 2014.

[69] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[70] I. van der Hoog, M. van de Kerkhof, M. van Kreveld, M. Löffler, F. Staals, J. Urhausen, and J. L. Vermeulen. Mapping multiple regions to the grid with bounded Hausdorff distance. In *Workshop on Algorithms and Data Structures*, pages 627–640. Springer, 2021.

[71] Z. Huo, X. Meng, H. Hu, and Y. Huang. You can walk alone: trajectory privacy-preserving through significant stays protection. In *International conference on database systems for advanced applications*, pages 351–366. Springer, 2012.

[72] H. Imai and M. Iri. An optimal algorithm for approximating a piecewise linear function. *Journal of Information Processing*, 9(3):159–162, 1986.

[73] H. Imai and M. Iri. Polygonal approximations of a curve—formulations and algorithms. *Computational Morphology*, pages 71–86, 1988.

[74] C. S. Jensen, H. Lu, and B. Yang. Indexing the trajectories of moving objects in symbolic indoor space. In *International Symposium on Spatial and Temporal Databases*, pages 208–227. Springer, 2009.

[75] H. Jeung, H. Lu, S. Sathe, and M. L. Yiu. Managing evolving uncertainty in trajectory databases. *IEEE Transactions on Knowledge and Data Engineering*, 26(7):1692–1705, 2013.

[76] B. Jiang and C. Claramunt. A structural approach to the model generalization of an urban street network. *GeoInformatica*, 8(2):157–171, 2004.

[77] B. Jiang and C. Claramunt. Topological analysis of urban street networks. *Environment and Planning B: Planning and design*, 31(1):151–162, 2004.

[78] B. Jiang and L. Harrie. Selection of streets from a network using self-organizing maps. *Transactions in GIS*, 8(3):335–350, 2004.

[79] P. Jin, J. Du, C. Huang, S. Wan, and L. Yue. Detecting hotspots from trajectory data in indoor spaces. In *International Conference on Database Systems for Advanced Applications*, pages 209–225. Springer, 2015.

[80] L. Johard and E. Ruffaldi. Self-organizing trajectories. *Pattern Recognition Letters*, 84:177–184, 2016.

[81] S. Karagiorgou, D. Pfoser, and D. Skoutas. Segmentation-based road network construction. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 460–463, 2013.

[82] T. Kashiyama, Y. Pang, and Y. Sekimoto. Open PFLOW: Creation and evaluation of an open dataset for typical people mass movement in urban areas. *Transportation research part C: emerging technologies*, 85:249–267, 2017.

[83] V. Keikha, M. van de Kerkhof, M. van Kreveld, I. Kostitsyna, M. Löffler, F. Staals, J. Urhausen, J. L. Vermeulen, and L. Wiratma. Convex partial transversals of planar regions. In *29th International Symposium on Algorithms and Computation, ISAAC 2018*, pages 52:1–52:12. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2018.

[84] M. van de Kerkhof, T. de Jong, R. Parment, M. Löffler, A. Vaxman, and M. van Kreveld. Design and automated generation of Japanese picture puzzles. *Computer Graphics Forum*, 38(2):343–353, 2019.

[85] M. van de Kerkhof, I. Kostitsyna, and M. Löffler. Embedding ray intersection graphs and global curve simplification. *arXiv preprint arXiv:2109.00042*, 2021.

[86] M. van de Kerkhof, I. Kostitsyna, M. Löffler, M. Mirzanezhad, and C. Wenk. Global curve simplification. In *27th Annual European Symposium on Algorithms (ESA 2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2019.

[87] M. van de Kerkhof, I. Kostitsyna, M. van Kreveld, M. Löffler, and T. Ophelders. Route-preserving road network generalization. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, pages 381–384, 2020.

[88] I. Kostitsyna, M. Löffler, V. Polishchuk, and F. Staals. On the complexity of minimum-link path problems. *Journal of Computational Geometry*, 8(2):80–108, 2017.

[89] M. van Kreveld, M. Löffler, and L. Wiratma. On optimal polyline simplification using the Hausdorff and Fréchet distance. In *Proc. 34th International Symposium on Computational Geometry (SoCG)*, volume 56, pages 1–14, 2018.

[90] M. van Kreveld, M. Löffler, and F. Staals. Central trajectories. *arXiv preprint arXiv:1501.01822*, 2015.

[91] M. van Kreveld and J. Peschier. On the automated generalization of road network maps. In *Proc. 3rd International Conference in GeoComputation*, 1998.

[92] J.-G. Lee, J. Han, and X. Li. Trajectory outlier detection: A partition-and-detect framework. In *2008 IEEE 24th International Conference on Data Engineering*, pages 140–149. IEEE, 2008.

[93] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 593–604. ACM, 2007.

[94] W.-C. Lee and J. Krumm. Trajectory preprocessing. In *Computing with Spatial Trajectories*, pages 3–33. Springer, 2011.

[95] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu, and W.-Y. Ma. Mining user similarity based on location history. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, pages 1–10, 2008.

[96] X. Li, J. Han, S. Kim, and H. Gonzalez. Roam: Rule-and motif-based anomaly detection in massive moving object data sets. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, pages 273–284, 2007.

[97] X. Li, J. Han, J.-G. Lee, and H. Gonzalez. Traffic density-based discovery of hot routes in road networks. In *International Symposium on Spatial and Temporal Databases*, pages 441–459. Springer, 2007.

[98] X. Lin, S. Ma, H. Zhang, T. Wo, and J. Huai. One-pass error bounded trajectory simplification. *arXiv preprint arXiv:1702.05597*, 2017.

[99] X. Liu, T. Ai, and Y. Liu. Road density analysis based on skeleton partitioning for road generalization. *Geo-spatial Information Science*, 12(2):110–116, 2009.

[100] X. Liu, H. Chen, and C. Andris. TrajGANs: Using generative adversarial networks for geo-privacy protection of trajectory data (vision paper). In *Location Privacy and Security Workshop*, pages 1–7, 2018.

[101] X. Liu, F. B. Zhan, and T. Ai. Road selection based on Voronoi diagrams and "strokes" in map generalization. *International Journal of Applied Earth Observation and Geoinformation*, 12:S194–S202, 2010.

[102] J. A. Long. Kinematic interpolation of movement data. *International Journal of Geographical Information Science*, 30(5):854–868, 2016.

[103] W. Mackaness. Analysis of urban road networks to support cartographic generalization. *Cartography and Geographic Information Systems*, 22(4):306–316, 1995.

[104] W. A. Mackaness and K. M. Beard. Use of graph theory to support map generalization. *Cartography and Geographic Information Systems*, 20(4):210–221, 1993.

[105] A. Makris, C. L. da Silva, V. Bogorny, L. O. Alvares, J. A. Macedo, and K. Tserpes. Evaluating the effect of compressing algorithms for trajectory similarity and classification problems. *GeoInformatica*, pages 1–33, 2021.

[106] N. Meratnia and A. Rolf. Spatiotemporal compression techniques for moving point objects. In *International Conference on Extending Database Technology*, pages 765–782. Springer, 2004.

[107] W. Meulemans. Map matching with simplicity constraints. *arXiv preprint arXiv:1306.2827*, 2013.

[108] H. J. Miller and M. F. Goodchild. Data-driven geography. *GeoJournal*, 80(4):449–461, 2015.

[109] M. Montanino and V. Punzo. Trajectory data reconstruction and simulation-based validation against macroscopic traffic patterns. *Transportation Research Part B: Methodological*, 80:82–106, 2015.

[110] B. Morisset and A. Ruas. Simulation and agent modelling for road selection in generalisation. In *Proc. ICA 18th International Cartographic Conference*, pages 1376–1380, 1997.

[111] M. E. Nergiz, M. Atzori, and Y. Saygin. Towards trajectory anonymization: a generalization-based approach. In *Proceedings of the SIGSPATIAL ACM GIS 2008 International Workshop on Security and Privacy in GIS and LBS*, pages 52–61. ACM, 2008.

[112] P. Newson and J. Krumm. Hidden Markov map matching through noise and sparseness. In *Proc. 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2009.

[113] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[114] M. Notaro, F. Alkolibi, E. Fadda, and F. Bakhrjy. Trajectory analysis of Saudi Arabian dust storms. *Journal of Geophysical Research: Atmospheres*, 118(12):6028–6043, 2013.

[115] OpenStreetMap (OSM). https://www.openstreetmap.org.

[116] M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4):168–173, 1981.

[117] N. Pelekis, I. Kopanakis, E. Kotsifakos, E. Frentzos, and Y. Theodoridis. Clustering trajectories of moving objects in an uncertain world. In *2009 Ninth IEEE International Conference on Data Mining*, pages 417–427. IEEE, 2009.

[118] T. Prentow, A. Thom, H. Blunck, and J. Vahrenhold. Making sense of trajectory data in indoor spaces. In *2015 16th IEEE International Conference on Mobile Data Management*, volume 1, pages 116–121. IEEE, 2015.

[119] V. Punzo, M. T. Borzacchiello, and B. Ciuffo. On the assessment of vehicle trajectory data accuracy and application to the Next Generation SIMulation (NGSIM) program data. *Transportation Research Part C: Emerging Technologies*, 19(6):1243–1262, 2011.

[120] P. Samarati and L. Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report, SRI International, 1998.

[121] M. Schaefer. Complexity of some geometric and topological problems. In *International Symposium on Graph Drawing*, pages 334–344. Springer, 2009.

[122] W. Shi, S. Shen, and Y. Liu. Automatic generation of road network map from massive GPS, vehicle trajectories. In *2009 12th International IEEE Conference on Intelligent Transportation Systems*, pages 1–6. IEEE, 2009.

[123] R. Šuba, M. Meijers, and P. V. Oosterom. Continuous road network generalization throughout all scales. *ISPRS International Journal of Geo-Information*, 5(8):145, 2016.

[124] Y. Tao, A. Both, R. I. Silveira, K. Buchin, S. Sijben, R. S. Purves, P. Laube, D. Peng, K. Toohey, and M. Duckham. A comparative analysis of trajectory similarity measures. *GIScience & Remote Sensing*, 0(0):1–27, 2021.

[125] R. C. Thomson and R. Brooks. Exploiting perceptual grouping for map analysis, understanding and generalization: The case of road and river networks. In *International Workshop on Graphics Recognition*, pages 148–157, 2001.

[126] R. C. Thomson and D. E. Richardson. A graph theory approach to road network generalisation. In *Proc. 17th International Cartographic Conference*, pages 1871–1880, 1995.

[127] R. C. Thomson and D. E. Richardson. The 'good continuation' principle of perceptual organization applied to the generalization of road networks. In *Proc. of the 19th International Cartographic Conference*, pages 1215—-1223, 1999.

[128] M. R. Uddin, C. Ravishankar, and V. J. Tsotras. Finding regions of interest from trajectory data. In *2011 IEEE 12th International Conference on Mobile Data Management*, volume 1, pages 39–48. IEEE, 2011.

[129] N. Venkataramanan and A. Shriram. *Data privacy: principles and practice*. Chapman and Hall/CRC, 2016.

[130] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *Proceddings of the 18th International Conference on Data Engineering*, pages 673–684, 2002.

[131] R. Weiss and R. Weibel. Road network selection for small-scale maps using an improved centrality-based algorithm. *Journal of Spatial Information Science*, 2014(9):71–99, 2014.

[132] W. Yu, Y. Zhang, T. Ai, Q. Guan, Z. Chen, and H. Li. Road network generalization considering traffic flow patterns. *International Journal of Geographical Information Science*, 34(1):119–149, 2020.

[133] G. Yuan, S. Xia, L. Zhang, Y. Zhou, and C. Ji. Trajectory outlier detection algorithm based on structural features. *Journal of Computational Information Systems*, 7(11):4137–4144, 2011.

[134] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 316–324, 2011.

[135] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*, pages 99–108, 2010.

[136] Q. Zhang. Modeling structure and patterns in road network generalization. In *ICA Workshop on Generalisation and Multiple Representation*, 2004.

[137] Q. Zhang. Road network generalization based on connection analysis. In *Developments in Spatial Data Handling*, pages 343–353. Springer, 2005.

[138] Z. Zhang, K. Huang, and T. Tan. Comparison of similarity measures for trajectory clustering in outdoor surveillance scenes. In *ICPR (3)*, pages 1135–1138. Citeseer, 2006.

[139] Y. Zheng. Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology*, 6(3):29, 2015.

[140] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.-Y. Ma. Understanding mobility based on GPS data. In *Proceedings of the 10th International Conference on Ubiquitous Computing*, pages 312–321, 2008.

[141] Y. Zheng, X. Xie, and W.-Y. Ma. GeoLife: A collaborative social networking service among user, location and trajectory. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 33(2):32–39, 2010.

[142] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from GPS trajectories. In *Proceedings of the 18th International Conference on World Wide Web*, pages 791–800, 2009.

[143] Q. Zhou and Z. Li. Evaluation of properties to determine the importance of individual roads for map generalization. In *Advances in Cartography and GIScience*, volume 1, pages 459–475. Springer, 2011.