WALKING THE DOG FAST IN PRACTICE: ALGORITHM ENGINEERING OF THE FRÉCHET DISTANCE

Karl Bringmann,[*] *Marvin Künnemann,*[†] *and André Nusser*[‡]

ABSTRACT. The Fréchet distance provides a natural and intuitive measure for the popular task of computing the similarity of two (polygonal) curves. While a simple algorithm computes it in near-quadratic time, a strongly subquadratic algorithm cannot exist unless the Strong Exponential Time Hypothesis fails. Still, fast practical implementations of the Fréchet distance, in particular for *realistic* input curves, are highly desirable. This has even lead to a designated competition, the ACM SIGSPATIAL GIS Cup 2017: Here, the challenge was to implement a near-neighbors data structure under the Fréchet distance. The bottleneck of the top three implementations turned out to be precisely the decision procedure for the Fréchet distance.

In this work, we present a fast, certifying implementation for deciding the Fréchet distance, in order to (1) complement its pessimistic worst-case hardness by an empirical analysis on realistic input data and to (2) improve the state of the art for the GIS Cup challenge. We experimentally evaluate our implementation on a large benchmark consisting of several data sets (including handwritten characters and GPS trajectories). Compared to the winning implementation of the GIS Cup, we obtain running time improvements of up to more than two orders of magnitude for the decision procedure and of up to a factor of 30 for queries to the near-neighbors data structure.

1 Introduction

A variety of practical applications analyze and process trajectory data coming from different sources like GPS measurements, digitized handwriting, motion capturing, and many more. One elementary task on trajectories is to compare them, for example in the context of signature verification [33], map matching [17, 32, 19, 11], and clustering [13, 15]. In this work we consider the Fréchet distance as curve similarity measure as it is arguably the most natural and popular one. Intuitively, the Fréchet distance between two curves is explained using the following analogy. A person walks a dog, connected by a leash. Both walk along their respective curve, with possibly varying speeds and without ever walking backwards.

Over all such traversals, we search for the ones which minimize the leash length, i.e., we minimize the maximal distance the dog and the person have during the traversal.

Initially defined more than one hundred years ago [24], the Fréchet distance quickly gained popularity in computer science after the first algorithm to compute it was presented by Alt and Godau [2]. In particular, they showed how to decide whether two length-n curves have Fréchet distance at most δ in time $\mathcal{O}(n^2)$ by full exploration of a quadratic-sized search space, the so-called *free-space* (we refer to Section 3.1 for a definition). Almost twenty years later, it was shown that, conditional on the Strong Exponential Time Hypothesis (SETH), there cannot exist an algorithm with running time $\mathcal{O}(n^{2-\epsilon})$ for any $\epsilon > 0$ [7]. Even for realistic models of input curves, such as c-packed curves [21], exact computation of the Fréchet distance requires time $n^{2-o(1)}$ under SETH [7]. Only if we relax the goal to finding a $(1 + \epsilon)$-approximation of the Fréchet distance, algorithms with near-linear running times in n and c on c-packed curves are known to exist [21, 8].

It is a natural question whether these hardness results are mere theoretical worst-case results or whether computing the Fréchet distance is also hard in practice. This line of research was particularly fostered by the research community in form of the GIS Cup 2017 [31]. In this competition, the 28 contesting teams were challenged to give a fast implementation for the following problem: Given a data set of two-dimensional trajectories $\mathcal{D}$, answer queries that ask to return, given a curve π and query distance δ, all $\sigma \in \mathcal{D}$ with Fréchet distance at most δ to π. We call this the *near-neighbors problem*.

The three top implementations [6, 12, 22] use multiple layers of heuristic filters and spatial hashing to decide as early as possible whether a curve belongs to the output set or not, and finally use an essentially exhaustive Fréchet distance computation for the remaining cases. Specifically, these implementations perform the following steps:

(0) Preprocess $\mathcal{D}$.

On receiving a query with curve π and query distance δ:

(1) Use spatial hashing to identify candidate curves $\sigma \in \mathcal{D}$.

(2) For each candidate σ, decide whether π, σ have Fréchet distance $\leq \delta$:

  a) Use heuristics (*filters*) for a quick resolution in simple cases.
  b) If unsuccessful, use a *complete decision procedure via free-space exploration*.

Let us highlight the *Fréchet decider* outlined in steps 2a and 2b: Here, *filters* refer to sound, but incomplete Fréchet distance decision procedures, i.e., whenever they succeed to find an answer, they are correct, but they may return that the answer remains unknown. In contrast, a *complete decision procedure via free-space exploration* explores a sufficient part of the free space (the search space) to always determine the correct answer. As it turns out, the bottleneck in all three implementations is precisely Step 2b, the complete decision procedure via free-space exploration. Especially [6] improved upon the naive implementation of the free-space exploration by designing very basic pruning rules, which might be the advantage because of which they won the competition. There are two directions for further substantial

improvements over the cup implementations: (1) increasing the range of instances covered by fast filters and (2) algorithmic improvements of the exploration of the reachable free-space.

Our Contribution. We develop a fast, practical Fréchet distance implementation. To this end, we give a complete decision procedure via free-space exploration that uses a divide-and-conquer interpretation of the Alt-Godau algorithm for the Fréchet distance and optimize it using sophisticated pruning rules. These pruning rules greatly reduce the search space for the realistic benchmark sets we consider – this is surprising given that simple constructions generate hard instances which require the exploration of essentially the full quadratic-sized search space [7, 9]. Furthermore, we present improved filters that are sufficiently fast compared to the complete decider. Here, the idea is to use adaptive step sizes (combined with useful heuristic tests) to achieve essentially "sublinear" time behavior for testing if an instance can be resolved quickly. Additionally, our implementation is certifying (see [27] for a survey on certifying algorithms), meaning that for every decision of curves being close/far, we provide a short proof (certificate) that can be checked easily; we also implemented a computational check of these certificates. See Section 8 for details.

An additional contribution of this work is the creation of benchmarks to make future implementations more easily comparable. We compile benchmarks both for the near-neighbors problem (Steps 0 to 2) and for the decision problem (Step 2). For this, we used publicly available curve data and created queries in a way that should be representative for the performance analysis of an implementation. As data sets we use the GIS Cup trajectories [29], a set of handwritten characters called the Character Trajectories Data Set [18] from [20], and the GeoLife data set [25] of Microsoft Research [35, 34, 36]. Our benchmarks cover different distances and also curves of different similarity, giving a broad overview over different settings. We make the source code as well as the benchmarks publicly available to enable independent comparisons with our approach.[1] Additionally, we particularly focus on making our implementation easily readable to enable and encourage others to reuse the code.

Evaluation. The GIS Cup 2017 had 28 submissions, with the top three submissions[2] (in decreasing order) due to Bringmann and Baldus [6], Buchin et al. [12], and Dütsch and Vahrenhold [22]. We compare our implementation with all of them by running their implementations on our new benchmark set for the near-neighbors problem and also comparing to the improved decider of [6]. The comparison shows significant speed-ups up to almost a factor of 30 for the near-neighbors problem and up to more than two orders of magnitude for the decider.

---

[1]Code and benchmarks are available at https://gitlab.mpi-klsb.mpg.de/anusser/frechet_distance.

[2]The submissions were evaluated "for their correctness and average performance on a[sic!] various large trajectory databases and queries". Additional criteria were the following: "We will use the total elapsed wall clock time as a measure of performance. For breaking ties, we will first look into the scalability behavior for more and more queries on larger and larger datasets. Finally, we break ties on code stability, quality, and readability and by using different datasets."

Related Work. The best known algorithm for deciding the Fréchet distance runs in time $O(n^2 \frac{(\log \log n)^2}{\log n})$ on the word RAM [10]. This relies on the Four Russians technique and is mostly of theoretical interest. In light of the quadratic-time conditional lower bound in the worst case [7], it is not surprising that faster algorithms are known only for some restrictive classes of curves such as closed convex curves [3] and curves consisting of large segments (compared to the Fréchet distance) [26]. If we additionally relax the goal to *approximating* the Fréchet distance, strongly subquadratic algorithm are known for realistic input models including κ-bounded, κ-straight, and c-packed curves [3, 21, 8]. Unfortunately, the algorithmic techniques used for these results crucially rely on the input restrictions and/or are inherently approximative rather than exact. Further results are known for variants of the Fréchet distance, e.g., the discrete Fréchet distance [1, 23, 4].

After the GIS Cup 2017, several practical papers studying aspects of the Fréchet distance appeared [5, 16, 30]. Some of this work [5, 16] addressed how to improve upon the spatial hashing step (Step 1) if we relax the requirement of exactness. Since this is orthogonal to our approach of improving the complete decider, these improvements could possibly be combined with our algorithm. The other work [30] neither compared with the GIS Cup implementations, nor provided their source code publicly to allow for a comparison, which is why we have to ignore it here.

Organization. First, in Section 2, we present all the core definitions. Subsequently, we explain our complete decider in Section 3. The following section then explains the decider and its filtering steps. Then, in Section 5, we present a query data structure which enables us to compare to the GIS Cup submissions. Section 6 contains some details regarding the implementation to highlight crucial points that are relevant for similar implementations. We conduct extensive experiments in Section 7, detailing the improvements over the current state of the art by our implementation. Finally, in Section 8, we describe how we make our implementation certifying and evaluate the certifying code experimentally.

2 Preliminaries

Our implementation as well as the description are restricted to two dimensions, however, the approach can also be generalized to polygonal curves in d dimensions. Therefore, a *curve* π is defined by its *vertices* $\pi_1, \ldots, \pi_n \in \mathbb{R}^2$ which are connected by straight lines. We also allow continuous indices as follows. For $p = i + \lambda$ with $i \in \{1, \ldots, n\}$ and $\lambda \in [0, 1]$, let

$$\pi_p := (1 - \lambda)\pi_i + \lambda\pi_{i+1}.$$

We call the π_p with $p \in [1, n]$ the *points* on π. A subcurve of π which starts at point p and ends at point q on π is denoted by $\pi_{p \ldots q}$. In the remainder, we denote the *number of vertices* of π (resp. σ) with n (resp. m) if not stated otherwise. We denote the length of a curve π by $\|\pi\|$, i.e., the sum of the Euclidean lengths of its line segments. Additionally, we use $\|v\|$ for the Euclidean norm of a vector $v \in \mathbb{R}^2$. For two curves π and σ, the *Fréchet distance* $d_F(\pi, \sigma)$ is defined as

$$d_F(\pi, \sigma) := \inf_{\substack{f \in \mathcal{T}_n \\ g \in \mathcal{T}_m}} \max_{t \in [0,1]} \left\| \pi_{f(t)} - \sigma_{g(t)} \right\|,$$

where $\mathcal{T}_k$ is the set of monotonically increasing continuous surjective functions $f : [0,1] \to [1,k]$. We define a *traversal* as a pair $(f,g) \in \mathcal{T}_n \times \mathcal{T}_m$. Given two curves π, σ and a query distance δ, we call them *close* if $d_F(\pi,\sigma) \le \delta$ and *far* otherwise. There are two problem settings that we consider in this paper:

Decider Setting: Given curves π, σ and a distance δ, decide whether $d_F(\pi,\sigma) \le \delta$. (With such a decider, we can compute the exact distance by using parametric search in theory and binary search in practice.)

Query Setting: Given a curve dataset $\mathcal{D}$, build a data structure that on query (π, δ) returns all $\sigma \in \mathcal{D}$ with $d_F(\pi,\sigma) \le \delta$.

We mainly focus on the decider in this work. To allow for a comparison with previous implementations (which are all in the query setting), we also run experiments with our decider plugged into a data structure for the query setting.

2.1 Preprocessing

When reading the input curves we immediately compute additional data which is stored with each curve:

Prefix Distances: To be able to quickly compute the curve length between any two vertices of π, we precompute the prefix lengths, i.e., the curve lengths $\|\pi_{1...i}\|$ for every $i \in \{2,\dots,n\}$. We can then compute the curve length for two indices $i < i'$ on π by $\|\pi_{i...i'}\| = \|\pi_{1...i'}\| - \|\pi_{1...i}\|$.

Bounding Box: We compute the bounding box of all curves, which is a simple coordinate-wise maximum and minimum computation.

Both of these preprocessing steps are extremely cheap as they only require a single pass over all curves, which we anyway do when parsing them. In the remainder of this work we assume that this additional data was already computed, in particular, we do not measure it in our experiments as it is dominated by reading the curves.

3 Complete Decider

The key improvement of this work lies in the complete decider via free-space exploration. Here, we use a divide-and-conquer interpretation of the algorithm of Alt and Godau [2] which is similar to [6] where a free-space diagram is built recursively. This interpretation allows us to prune away large parts of the search space by designing powerful *pruning rules* identifying parts of the search space that are irrelevant for determining the correct output. Before describing the details, we formally define the free-space diagram.
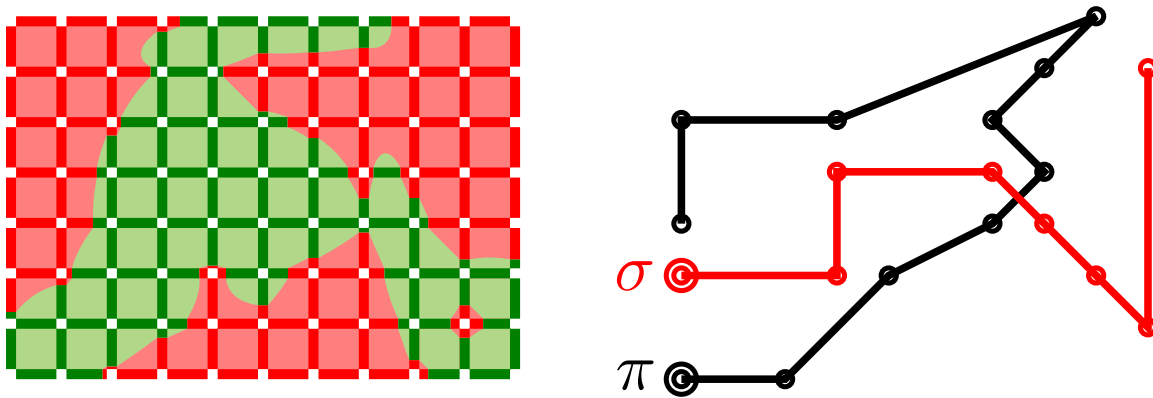
Figure 1: Example of a free-space diagram for curves π (black) and σ (red). Curve π is on the horizontal axis of the free-space diagram, while σ is on the vertical axis; we use this convention in the remainder. The doubly-circled vertices mark the start. The free-space, i.e., the pairs of indices of points which are close, is colored green. The non-free areas are colored red. The threshold distance δ is roughly the distance between the first vertex of σ and the third vertex of π.

3.1   Free-Space Diagram

The free-space diagram was first defined in [2]. Given two polygonal curves π and σ and a distance δ, it is defined as the set of all pairs of indices of points from π and σ that are close to each other, i.e.,

$$F := \{(p, q) \in [1, n] \times [1, m] \mid \|\pi_p - \sigma_q\| \leq \delta\}.$$

For an example see Figure 1. A *path* from a to b in the free-space diagram F is defined as a continuous mapping $P : [0, 1] \to F$ with $P(0) = a$ and $P(1) = b$. A path P in the free-space diagram is *monotone* if $P(x)$ is component-wise at most $P(y)$ for any $0 \leq x \leq y \leq 1$. The *reachable space* is then defined as

$$R := \{(p, q) \in F \mid \text{there exists a monotone path from } (1, 1) \text{ to } (p, q) \text{ in } F\}.$$

Figure 2 shows the reachable space for the free-space diagram of Figure 1. It is well known that $d_F(\pi, \sigma) \leq \delta$ if and only if $(n, m) \in R$.

This leads us to a simple dynamic programming algorithm to decide whether the Fréchet distance of two curves is at most some threshold distance. We iteratively compute R starting from $(1, 1)$ and ending at (n, m), using the previously computed values. As R is potentially a set of infinite size, we have to discretize it. A natural choice is to restrict to cells. The *cell* of R with coordinates $(i, j) \in \{1, \ldots, n-1\} \times \{1, \ldots, m-1\}$ is defined as $C_{i,j} := [i, i+1] \times [j, j+1]$. This is a natural choice as given $R \cap C_{i-1,j}$ and $R \cap C_{i,j-1}$, we can compute $R \cap C_{i,j}$ in constant time; this follows from the simple fact that $F \cap C_{i,j}$ is convex [2]. We call this computation of the outputs of a cell the *cell propagation*. This algorithm runs in time $\mathcal{O}(nm)$ and was introduced by Alt and Godau [2].
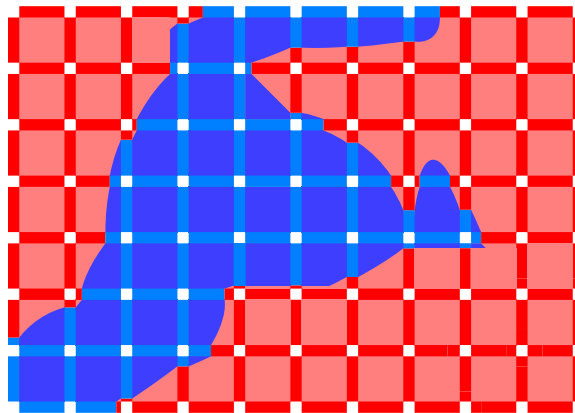
Figure 2: Reachable space of the free-space diagram in Figure 1. The reachable part is blue and the non-reachable part is red. Note that the reachable part is a subset of the free-space. We use this color scheme in the remainder of this work.

3.2 Basic Algorithm

For integers $1 \leq i \leq i' \leq n, 1 \leq j \leq j' \leq m$ we call the set $B = [i, i'] \times [j, j']$ a *box*. We denote the left/right/bottom/top *boundaries* of B by $B_l = \{i\} \times [j, j'], B_r = \{i'\} \times [j, j'], B_b = [i, i'] \times \{j\}, B_t = [i, i'] \times \{j'\}$. The *left input* of B is $B_l^R = B_l \cap R$, and its *bottom input* is $B_b^R = B_b \cap R$. Similarly, the *right/top output* of B is $B_r^R = B_r \cap R$, $B_t^R = B_t \cap R$. A box is a *cell* if $i + 1 = i'$ and $j + 1 = j'$. We always denote the lower left corner of a box by (i, j) and the top right by (i', j'), if not mentioned otherwise.

A recursive variant of the standard free-space decision procedure is as follows: Start with $B = [1, n] \times [1, m]$. At any recursive call, if B is a cell, then determine its outputs from its inputs in constant time, as described by [2]. Otherwise, split B vertically or horizontally into B_1, B_2 and first compute the outputs of B_1 from the inputs of B and then compute the outputs of B_2 from the inputs of B and the outputs of B_1. In the end, we just have to check $(n, m) \in R$ to decide whether the curves are close or far. This is a constant-time operation after calculating all outputs.

Now comes the main idea of our approach: we try to avoid recursive splitting by directly computing the outputs for non-cell boxes using certain rules. We call them *pruning rules* as they enable pruning large parts of the recursion tree induced by the divide-and-conquer approach. Our pruning rules are heuristic, meaning that they are not always applicable, however, we show in the experiments that on practical curves they apply very often and therefore massively reduce the number of recursive calls. The detailed pruning rules are described in Section 3.3. Using these rules, we change the above recursive algorithm as follows. In any recursive call on box B, we first try to apply the pruning rules. If this is successful, then we obtained the outputs of B and we are done with this recursive call. Otherwise, we perform the usual recursive splitting. Corresponding pseudocode is shown in Algorithm 1.

In the remainder of this section, we describe our pruning rules and their effects.

**Algorithm 1** Recursive Decider of the Fréchet Distance

---

1: **procedure** $\text{DECIDEFRÉCHETDISTANCE}(\pi, \sigma, \delta)$
2:     $\text{COMPUTEOUTPUTS}(\pi, \sigma, [1, n] \times [1, m])$
3:     **return** $[(n, m) \in R]$

4: **procedure** $\text{COMPUTEOUTPUTS}(\pi, \sigma, B = [i, i'] \times [j, j'], \delta)$
5:     **if** B is a cell **then**
6:         compute outputs by cell propagation
7:     **else**
8:         use pruning rules I to IV to compute outputs of B
9:         **if** not all outputs have been computed **then**
10:           **if** $j' - j > i' - i$ **then** $\triangleright$ split horizontally
11:              $B_1 = [i, i'] \times [j, \lfloor (j + j')/2 \rfloor]$
12:              $B_2 = [i, i'] \times [\lfloor (j + j')/2 \rfloor, j']$
13:           **else** $\triangleright$ split vertically
14:              $B_1 = [i, \lfloor (i + i')/2 \rfloor] \times [j, j']$
15:              $B_2 = [\lfloor (i + i')/2 \rfloor, i'] \times [j, j']$
16:           $\text{COMPUTEOUTPUTS}(\pi, \sigma, B_1)$
17:           $\text{COMPUTEOUTPUTS}(\pi, \sigma, B_2)$
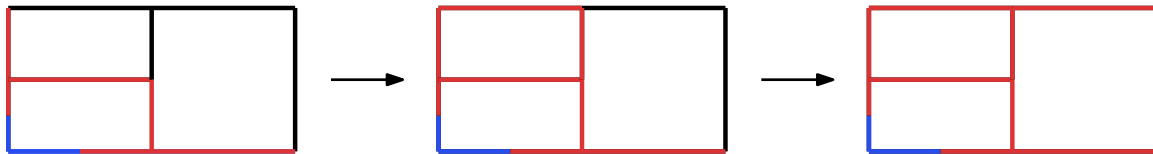
Figure 3: Output computation of a box when inputs are empty. First we can compute the outputs of the top left box and then the outputs of the right box. In this example, we then know that the curves have a Fréchet distance greater than δ as (n, m) is not reachable.

3.3 Pruning Rules

In this section we introduce the rules that we use to compute outputs of boxes which are above cell-level in certain special cases. Note that we aim at catching special cases which occur often in practice, as we cannot hope for improvements on adversarial instances due to the conditional lower bound of [7]. Therefore, we make no claims whether they are applicable, only that they are sound and fast. In what follows, we call a boundary *empty* if its intersection with R is $\emptyset$.

Rule I: Empty Inputs

The simplest case where we can compute the outputs of a box B is if both inputs are empty, i.e. $B_b^R = B_l^R = \emptyset$. In this case no propagation of reachability is possible and thus the outputs are empty as well, i.e. $B_t^R = B_r^R = \emptyset$. See Figure 3 for an example.
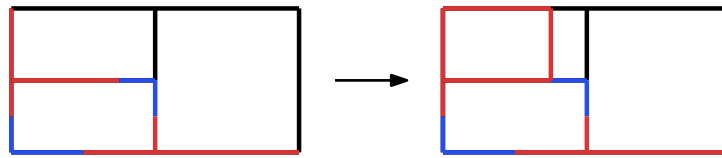
Figure 4: This is an example of shrinking a box in case one of the inputs is empty and the other one starts with an empty part. In this example the top left box has an empty input on the left and the start of the bottom input is empty as well. Thus, we can shrink the box to the right part.
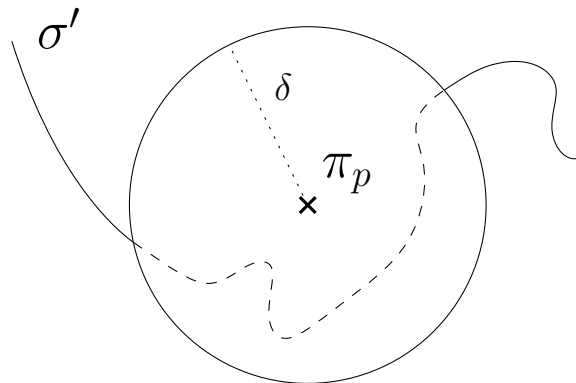
Figure 5: Example of a point π_p and a curve σ' which lead to a simple boundary.

Rule II: Shrink Box

Instead of directly computing the outputs, this rule allows us to shrink the box we are currently working on, which reduces the problem size. Assume that for a box B we have that $B_b^R = \emptyset$ and the lowest point of B_l^R is $(i, j_{\min})$ with $j_{\min} > j$. In this case, no pair in $[i, i'] \times [j, j_{\min}]$ is reachable. Thus, we can shrink the box to the coordinates $[i, i'] \times [\lfloor j_{\min} \rfloor, j']$ without losing any reachability information. An equivalent rule can be applied if we swap the role of B_b and B_l. See Figure 4 for an example of applying this rule.

Rule III: Simple Boundaries

Simple boundaries are boundaries of a box that contain at most one free component. To define this formally, a set $\mathcal{I} \subseteq [1, n] \times [1, m]$ is called an *interval* if $\mathcal{I} = \emptyset$ or $\mathcal{I} = \{p\} \times [q, q']$ or $\mathcal{I} = [q, q'] \times \{p\}$ for real p and an interval $[q, q']$. In particular, the four boundaries of a box $B = [i, i'] \times [j, j']$ are intervals. We say that an interval $\mathcal{I}$ is *simple* if $\mathcal{I} \cap F$ is again an interval. Geometrically, we have a free interval of a point π_p and a curve $\sigma_{q \dots q'}$ (which is the form of a boundary in the free-space diagram) if the circle of radius δ around π_p intersects $\sigma_{q \dots q'}$ at most twice. See Figure 5 for an example. We call such a boundary simple because it is of low complexity, which we can exploit for pruning.

There are three pruning rules that we do based on simple boundaries (see Figure 6 for visualizations). They are stated here for the top boundary B_t, but symmetric rules apply

a)   b)   c)
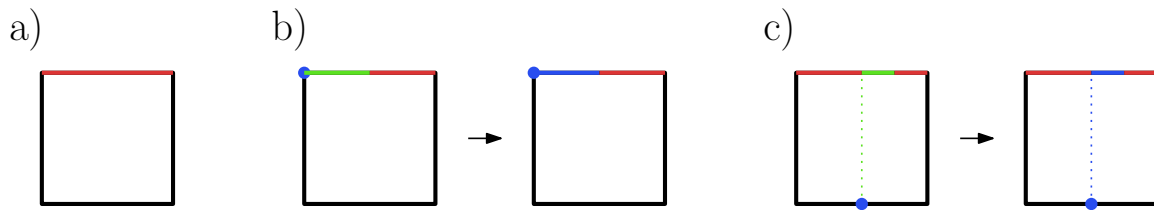
Figure 6: Visualization of the rules for computing outputs using simple boundaries. All three cases are visualized with the top boundary being simple. In a) the boundary is non-free and therefore no point on it can be reachable. In b) the boundary's beginning is free and reachable, enabling us to propagate the reachability to the entire free interval. In c) we can propagate the reachability of a point on the bottom boundary, using a free interval inside the box, to the beginning of the free interval of the top boundary and thus decide the entire boundary. The rules for the right boundary being simple are equivalent.

to B_r. Later, in Section 3.4, we then explain how to actually compute simple boundaries, i.e., also how to compute $B_t \cap F$. The pruning rules are:

(a) If B_t is simple because $B_t \cap F$ is empty then we also know that the output of this boundary is empty. Thus, we conclude that $B_t^R = \emptyset$ and we are done with B_t.

(b) Suppose that B_t is simple and, more specifically, of the form that it first has a free and then a non-free part; in other words, we have $(i, j') \in B_t \cap F$. Due to our recursive approach, we already computed the left inputs of the box and thus know whether the top left corner of the box is reachable, i.e. whether $(i, j') \in R$. If this is the case, then we also know the reachable part of our simple boundary: Since $(i, j') \in R$ and $B_t \cap F$ is an interval containing (i, j'), we conclude that $B_t^R = B_t \cap F$ and we are done with B_t.

(c) Suppose that B_t is simple, but the leftmost point $(i_{\min}, j')$ of $B_t \cap F$ has $i_{\min} > i$. In this case, we try to certify that $(i_{\min}, j') \in R$, because then it follows that $B_t^R = B_t \cap F$ and we are done with B_t. To check for reachability of $(i_{\min}, j')$, we try to propagate the reachability through the inside of the box, which in this case means to propagate it from the bottom boundary. We test whether $(i_{\min}, j)$ is in the input, i.e., if $(i_{\min}, j) \in B_b^R$, and whether $\{i_{\min}\} \times [j, j'] \subseteq F$ (by slightly modifying the algorithm for simple boundary computations). If this is the case, then we can reach every point in $B_t \cap F$ from $(i_{\min}, j)$ via $\{i_{\min}\} \times [j, j']$. Note that this is an operation in the complete decider where we explicitly use the *inside* of a box and not exclusively operate on its boundaries.

We also use symmetric rules by swapping "top" with "right" and "bottom" with "left".

Rule IV: Boxes at Free-Space Diagram Boundaries

The boundaries of a free-space diagram are a special form of boundary which allows us to introduce an additional rule. Consider a box B which touches the top boundary of
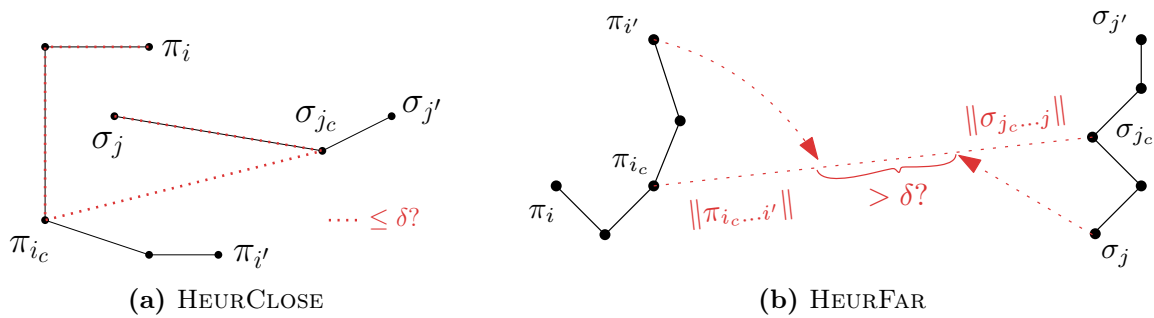
Figure 7: Visualizations of heuristic checks HeurClose and HeurFar.

the free-space diagram, i.e., $B = [i, i'] \times [j, m]$. Suppose the previous rules allowed us to determine the output for B_r^R. Since any valid traversal from $(1, 1)$ to (n, m) passing through B intersects B_r, the output B_t^R is not needed anymore, and we are done with B. A symmetric rule applies to boxes which touch the right boundary of the free-space diagram.

3.4   Implementation Details of Simple Boundaries

It remains to describe how we test whether a boundary is simple, and how we determine the free interval of a simple boundary. One important ingredient for the fast detection of simple boundaries are two simple heuristic checks that check whether two polygonal curves are close or far, respectively. The former check was already used in [6]. We first explain these heuristic checks, and then explain how to use them for the detection of simple boundaries.

Heuristic check whether two curves are close.   Given two subcurves $\pi' := \pi_{i\ldots i'}$ and $\sigma' := \sigma_{j\ldots j'}$, this filter heuristically tests whether $d_F(\pi', \sigma') \leq \delta$. Let $i_c := \lfloor \frac{i+i'}{2} \rfloor$ and $j_c := \lfloor \frac{j+j'}{2} \rfloor$ be the indices of the midpoints of π' and σ' (with respect to hops). Then $d_F(\pi', \sigma') \leq \delta$ holds if

$$\max\{\|\pi_{i\ldots i_c}\|, \|\pi_{i_c\ldots i'}\|\} + \|\pi_{i_c} - \sigma_{j_c}\| + \max\{\|\sigma_{j\ldots j_c}\|, \|\sigma_{j_c\ldots j'}\|\} \leq \delta.$$

The triangle inequality ensures that this is an upper bound on all distances between two points on the curves. For a visualization, see Figure 7a. Observe that all curve lengths that need to be computed in the above equation can be determined quickly due to our preprocessing, see Section 2.1. We call this procedure HeurClose(π', σ', δ).

Heuristic check whether two curves are far.   Symmetrically, we can test whether all pairs of points on π' and σ' are far by testing

$$\|\pi_{i_c} - \sigma_{j_c}\| - \max\{\|\pi_{i\ldots i_c}\|, \|\pi_{i_c\ldots i'}\|\} - \max\{\|\sigma_{j\ldots j_c}\|, \|\sigma_{j_c\ldots j'}\|\} > \delta.$$

We call this procedure HeurFar(π', σ', δ).

Algorithm 2 Checks if the boundary in the free-space diagram corresponding to $\{p\} \times [q, q']$ is simple.


1: **procedure** ISSIMPLEBOUNDARY($\pi_p, \sigma_{q...q'}$)
2:     **if** HEURFAR($\pi_p, \sigma_{q...q'}, \delta$) or HEURCLOSE($\pi_p, \sigma_{q...q'}, \delta$) **then**
3:         **return** "simple"

4:

5:     $C \leftarrow \begin{cases} \{\sigma_q\} & \text{, if } \|\pi_p - \sigma_q\| \leq \delta \\ \emptyset & \text{, otherwise} \end{cases}$ $\triangleright$ set of change points

6:     $s \leftarrow 1, j \leftarrow q$
7:     **while** $j < q'$ **do**
8:         **if** HEURCLOSE($\pi_p, \sigma_{j...j+s}, \delta$) **then**
9:             $j \leftarrow j + s$
10:            $s \leftarrow \min\{2s, q' - j\}$ $\triangleright$ double s but do not overstep q'
11:        **else if** HEURFAR($\pi_p, \sigma_{j...j+s}, \delta$) **then**
12:            $j \leftarrow j + s$
13:            $s \leftarrow \min\{2s, q' - j\}$ $\triangleright$ double s but do not overstep q'
14:        **else if** $s > 1$ **then**
15:            $s \leftarrow s/2$
16:        **else**
17:            $P \leftarrow \{j' \in (j, j+1] \mid \|\pi_p - \sigma_{j'}\| = \delta\}$
18:            $C \leftarrow C \cup P$
19:            $j \leftarrow j + 1$
20:            **if** $|C| > 2$ **then**
21:                **return** "not simple"

22:
23:     **return** "simple"

Computation of Simple Boundaries. Recall that an interval is defined as $I = \{p\} \times [q, q']$ (intervals of the form $[q, q'] \times \{p\}$ are handled symmetrically). The naive way to decide whether interval I is simple would be to go over all the segments of $\sigma_{q...q'}$ and compute the intersection with the circle of radius δ around π_p. However, this is too expensive because (i) computing the intersection of a disc and a segment involves taking a square root, which is an expensive operation with a large constant running time, and (ii) iterating over all segments of $\sigma_{q...q'}$ incurs a linear factor in n for large boxes, while we aim at a logarithmic dependence on n for simple boundary detection.

We avoid these issues by resolving long subcurves $\sigma_{j..j+s}$ using our heuristic checks (HEURCLOSE, HEURFAR). Here, s is an adaptive step size that grows whenever the heuristic checks were applicable, and shrinks otherwise. See Algorithm 2 for pseudocode of our simple boundary detection. It is straightforward to extend this algorithm to not only detect whether a boundary is simple, but also compute the free interval of a simple boundary; we call the resulting procedure SIMPLEBOUNDARY.
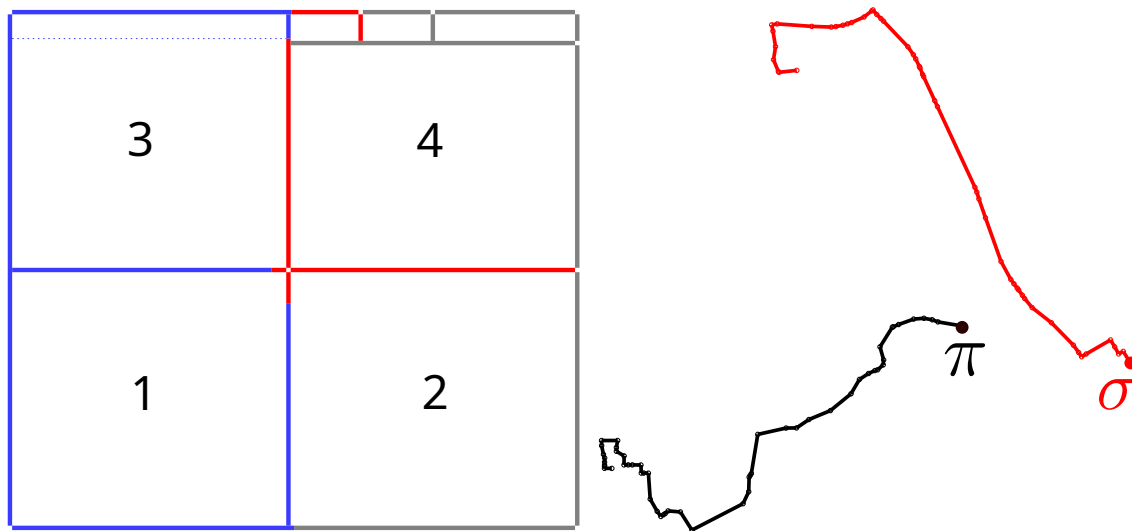
Figure 8: A free-space diagram as produced by our final implementation (left) with the corresponding curves (right). The beginnings of the curves are marked, and π is on the horizontal axis of the free-space diagram, while σ is on the vertical axis. The curves are taken from the SIGSPATIAL dataset. We number the boxes in the third level of the recursion from 1 to 4.

3.5   Effects of Combined Pruning Rules

All the pruning rules presented above can in practice lead to a reduction of the number of boxes that are necessary to decide the Fréchet distance of two curves. We exemplify this on two real-world curves; see Figure 8 on page 13 for the curves and their corresponding free-space diagram. We explain in the following where the single rules come into play. For *Box 1* we apply Rule IIIb twice – for the top and right output. The top boundary of *Box 2* is empty and thus we computed the outputs according to Rule IIIa. Note that the right boundary of this box is on the right boundary of the free-space diagram and thus we do not have to compute it according to Rule IV. For *Box 3* we again use Rule IIIb for the top, but we use Rule IIIc for the right boundary – the blue dotted line indicates that the reachability information is propagated through the box. For *Box 4* we first use Rule II to move the bottom boundary significantly up, until the end of the left empty part; we can do this because the bottom boundary is empty and the left boundary is simple, starting with an empty part. After two splits of the remaining box, we see that the two outputs of the leftmost box are empty as the top and right boundaries are non-free, using Rule IIIa. For the remaining two boxes we use Rule I as their inputs are empty.

This example illustrates how propagating through a box (in *Box 3*) and subsequently moving a boundary (in *Box 4*) leads to pruning large parts. Additionally, we can see how using simple boundaries leads to early decisions and thus avoids many recursive steps. In total, we can see how all the explained pruning rules together lead to a free-space diagram with only twelve boxes, i.e., twelve recursive calls, for curves with more than 50 vertices

---

Algorithm 3 High-level code of the Fréchet decider.

1: **procedure** $\text{DECIDER}(\pi, \sigma, \delta)$
2:    **if** start points π_1, σ_1 or end points π_n, σ_m are far **then return** "far"
3:    **for all** $f \in$ Filters **do**
4:        verdict $= f(\pi, \sigma, \delta)$
5:        **if** verdict $\in \{\text{"close", "far"}\}$ **then**
6:            **return** verdict
7:    **return** $\text{COMPLETEDECIDER}(\pi, \sigma, \delta)$

---

and more than 1500 reachable cells. Figure 9 shows what effects the pruning rules have by introducing them one by one in an example.

4   Decider with Filters

Now that we introduced the complete decider, we are ready to present the decider. We first give a high-level overview.
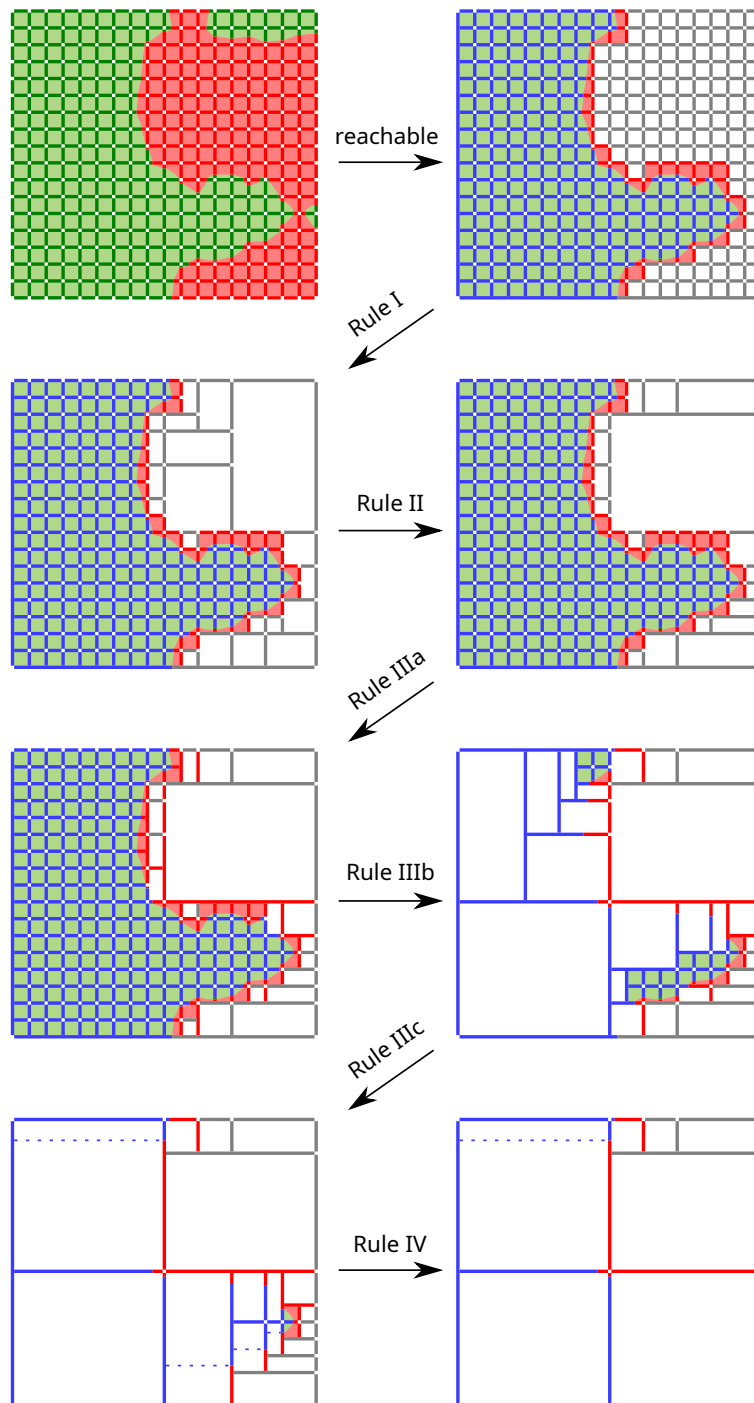
4.1   Decider

The decider can be divided into two parts:

1. Filters (see this section)

2. Complete decider via free-space exploration (see Section 3)

As outlined in Section 1, we first try to determine the correct output by using fast but incomplete filtering mechanisms and only resort to the slower complete decider presented in the last section if none of the heuristic deciders (*filters*) gave a result. The high-level pseudocode of the decider is shown in Algorithm 3.

The speed-ups introduced by our complete decider were already explained in Section 3. A second source for our speed-ups lies in the usage of a good set of filters. Interestingly, since our optimized complete decider via free-space exploration already solves many simple instances very efficiently, our filters have to be extremely fast to be useful – otherwise, the additional effort for an incomplete filter does not pay off. In particular, we cannot afford expensive preprocessing and ideally, we would like to achieve sublinear running times for our filters. To this end, we only use filters that can traverse large parts of the curves quickly. We achieve sublinear-type behavior by making previously used filters work with an adaptive step size (exploiting fast heuristic checks), and designing a new filter.

In the remainder of this section, we describe all the filters that we use to heuristically decide whether two curves are close or far. There are two types of filters: *positive* filters check whether a curve is close to the query curve and return either "close" or "unknown"; *negative* filters check if a curve is far from the query curve and return either "far" or "unknown".

Figure 9: A decider example introducing the pruning rules one by one. They are introduced from top to bottom and left to right. The arrows denote the rules which are introduced in between the two corresponding free-space diagrams. The curves of this example are shown in Figure 8.
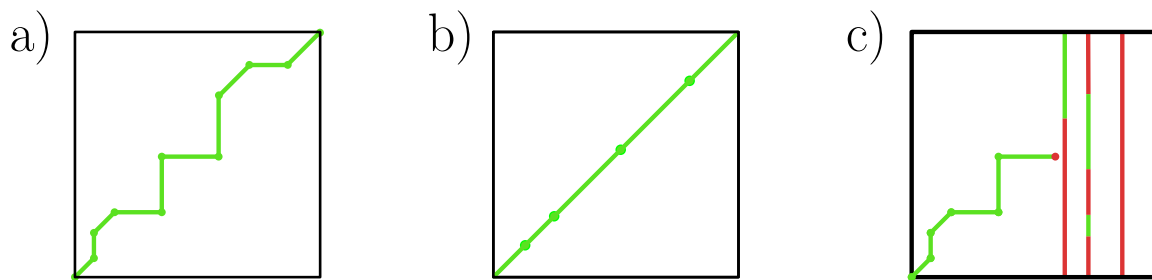
Figure 10: Sketches of the (a) greedy filter, (b) adaptive equal-time filter, and (c) negative filter. These sketches should be read as follows: the first dimension is the index on the first curve, while the second dimension is the index on the second curve. Recall that the green color indicates that the corresponding points are in distance at most δ; otherwise they are colored red. This visualization is similar to the free-space diagram.

4.2 Bounding Box Check

This is a positive filter already described in [22], which heuristically checks whether all pairs of points on π, σ are in distance at most δ. Recall that we compute the bounding box of each curve when we read it. We can thus check in constant time whether the furthest points on the bounding boxes of π, σ are in distance at most δ. If this is the case, then also all points of π, σ have to be close to each other and thus the free-space diagram is completely free and a valid traversal trivially exists.

4.3 Greedy

This is a positive filter. To assert that two curves π and σ are close, it suffices to find a traversal (f, g) satisfying $\max_{t \in [0,1]} \left\| \pi_{f(t)} - \sigma_{g(t)} \right\| \leq \delta$. We try to construct such a traversal staying within distance δ by making greedy steps that minimize the current distance. This may yield a valid traversal: if after at most $n + m$ steps we reach both endpoints and during the traversal the distance was always at most δ, we return "near". We can also get stuck: if a step on each of the curves would lead to a distance greater than δ, we return "unknown". A similar filter was already used in [6] and is a standard idea (see, e.g., [9, 26]), however, here we present a variant with adaptive step size. This means that instead of just advancing to the next node in the traversal, we try to make larger steps, leveraging the heuristic checks presented in Section 3.4. We adapt the step size depending on the success of the last step. For pseudocode of the greedy filter see Algorithm 4, and for a visualization see Figure 10a.

4.4 Adaptive Equal-Time

We also consider a variation of Greedy Filter, which we call Adaptive Equal-Time Filter. The only difference to Algorithm 4 is that the allowed steps are now:

$$S := \begin{cases} \{(i+1,j), (i,j+1), (i+1,j+1)\}, & \text{if } s = 1, \\ \left\{ \left(i+s, j + \left\lfloor \frac{m-j}{n-i} \cdot s \right\rfloor \right) \right\}, & \text{if } s > 1. \end{cases}$$

Algorithm 4 Greedy filter with adaptive step size.

1: **procedure** GREEDYFILTER(π, σ, δ)
2:      $i, j, s \leftarrow 1$
3:      **while** $i < n$ or $j < m$ **do**
4:          $S \leftarrow \begin{cases} \{(i+1, j), (i, j+1), (i+1, j+1)\}, & \text{if } s = 1 \\ \{(i+s, j), (i, j+s)\}, & \text{if } s > 1 \end{cases}$ ▷ possible steps
5:          $P \leftarrow \{(i', j') \in S \mid i' \leq n \text{ \& } j' \leq m \text{ \& } \textsc{HeurClose}(\pi_{i\ldots i'}, \sigma_{j\ldots j'}, \delta)\}$
6:          **if** $P = \emptyset$ **then**
7:              **if** $s = 1$ **then**
8:                  **return** "unknown"
9:              **else**
10:                 $s \leftarrow s/2$
11:         **else**
12:             $(i, j) \leftarrow \arg\min_{(i', j') \in P} \left\| \pi_{i'} - \sigma_{j'} \right\|$
13:             $s \leftarrow 2s$
14:     **return** "close"

In contrast to Greedy Filter, this searches for a traversal that stays as close as possible to the diagonal.

4.5   Negative

A negative filter was already used in [6] and [22]. However, changing this filter to use an adaptive step size does not seem to be practical when used with our approach. Preliminary tests showed that this filter would dominate our running time. Therefore, we developed a new negative filter which is more suited to be used with an adaptive step size and thus can be used with our approach.

Let (π_i, σ_j) be the points at which Greedy Filter got stuck. We check whether some point $\pi_{i+2^k}, k \in \mathbb{N}$, is far from all points of σ using HEURFAR. If so, we conclude that $d_F(\pi, \sigma) > \delta$. We do the same with the roles of π and σ exchanged. See Algorithm 5 for the pseudocode of this filter; for a visualization see Figure 10c.

5   Query Data Structure

In this section we give the details of extending the fast decider to compute the Fréchet distance in the query setting. Recall that in this setting we are given a *curve dataset* $\mathcal{D}$ that we want to preprocess for the following queries: Given a polygonal curve π (the *query curve*) and a threshold distance δ, report all $\sigma \in \mathcal{D}$ that are δ-close to π. To be able to compare our new approach to existing work (especially the submissions of the GIS Cup) we present a query data structure here, which is influenced by the one presented in [6].

The most important component that we need additionally to the decider to obtain an efficient query data structure is a mechanism to quickly determine a set of candidate

---

Algorithm 5 Negative filter, where in the two if statements we do a search with adaptive step size on σ and π, respectively.

1: **procedure** NEGATIVEFILTER(π, σ, δ)
2:      $(i, j) \leftarrow$ last indices of close points in greedy filter
3:      $s \leftarrow 1$
4:      **while** $i + s \leq n$ **do**
5:          **if** SIMPLEBOUNDARY($\pi_{i+s}, \sigma, \delta$) is non-free **then**
6:              **return** "far"
7:          $s \leftarrow 2s$
8:
9:      Repeat lines 3 to 7 with the roles of π and σ swapped
10:
11:     **return** "unknown"

Algorithm 6 The function for answering a range query.

1: **procedure** FINDCLOSECURVES(π, δ)
2:      $C \leftarrow$ KDTREE.QUERY(π, δ)
3:      $R \leftarrow \emptyset$
4:      **for all** $\sigma \in C$ **do**
5:          **if** FRECHETDISTANCEDECIDER(π, σ, δ) = "close" **then**
6:              $R \leftarrow R \cup \{\sigma\}$
7:      **return** R

---

curves on which we can then run the decider presented above. The candidate selection is done using a kd-tree on 8-dimensional points, similar to the octree used in [6], see 5.1 for more details. The high-level structure of the algorithm for answering queries is shown in Algorithm 6.

5.1 Kd-Tree

Fetching an initial set of candidate curves via a space-partitioning data structure was already used in [6, 12, 22]. We use a kd-tree which contains 8-dimensional points, each corresponding to one of the curves in the data set. Four dimensions of the 8-dimensional points are used for the start point and end point of the curve (two dimensions each). Note that two curves can only be close with respect to the Fréchet distance if their start points are close (and equivalently for the end points). Especially, if any of these four dimensions of two curves differ by more than δ, then these curves have a Fréchet distance larger than δ. The remaining four dimensions are used for the maximum/minimum coordinates in x/y direction. This is because, if the extremal coordinate in one direction of one curve compared to another is larger than δ, then the point that induces this large extremal coordinate cannot be matched to any point on the other curve. We can then query this kd-tree with the threshold distance δ and obtain a set of candidate curves. Note that this query does not have any false negatives, but might contain false positives, which we then filter out in the later stages of our algorithm.

---

6   Implementation Details

Square Root.   Computing which parts are close and which are far between a point and a segment involves intersecting a circle and a line segment, which in turn requires computing a square root. As square roots are computationally quite expensive, we avoid them by:

- filtering out simple comparisons by heuristic checks not involving square roots

- testing $x < a^2$ instead of $\sqrt{x} < a$ (and analogous for other comparisons)

While these changes seem trivial, they have a significant effect on the running time due to the large amount of distance computations in the implementation.

Recursion.   Note that the complete decider (Algorithm 1) is currently formulated as a recursive algorithm. Indeed, our implementation is also recursive, which is feasible due to the logarithmic depth of the recursion. An iterative variant that we implemented turned out to be equally fast but more complicated, thus we settled for the recursive variant.

7   Experiments

In the experiments, we aim to substantiate the following two claims. First, we want to verify that our main contribution, the decider, actually is a significant improvement over the state of the art. To this end, we compare our implementation with the – to our knowledge – currently fastest Fréchet distance decider, namely [6]. Second, we want to verify that our improvements in the decider setting also carry over to the query setting, also significantly improving the state of the art. To show this, we compare to the top three submissions of the GIS Cup.

We use three different data sets: the GIS Cup set (Sigspatial) [29], the handwritten characters (Characters) [18], and the GeoLife data set (GeoLife) [25]. For all experiments, we used a laptop with an Intel i5-6440HQ processor with 4 cores and 16GB of RAM.

Hypotheses.   In what follows, we verify the following hypotheses:

1. Our implementation is significantly faster than the fastest previously known implementation in the query and in the decider setting.

2. Our implementation is fast on a wide range of data sets.

3. Each of the described improvements of the decider speeds up the computation significantly.

4. The running time of the complete decider is proportional to the number of recursive calls.

Table 1: Information about the data sets used in the benchmarks.

data set	type	#curves	mean hops	stddev hops
SIGSPATIAL	synthetic GPS-like	20199	247.8	154.0
CHARACTERS	handwritten	2858	120.9	21.0
GEOLIFE	GPS (multi-modal)	16966	1080.4	1844.1

The first two we verify by running time comparisons on different data sets. The third we verify by leaving out single pruning rules and then comparing the running time with the final implementation. Finally, we verify the fourth hypothesis by correlating the running time for different decider computations against the number of recursive calls encountered during the computation.

7.1 Data Sets Information.

Some properties of the data sets are shown in Table 1. SIGSPATIAL has the most curves, while GEOLIFE has by far the longest. CHARACTERS is interesting as it does not stem from GPS data. By this selection of data sets, we hope to cover a sufficiently diverse set of curves.

Hardware. We used standard desktop hardware for our experiments. More specifically, we used a laptop with an Intel i5-6440HQ processor with 4 cores (2.6 to 3.1 GHz) with cache sizes 256KiB, 1MiB, and 6MiB (L1, L2, L3).

Code. The implementation is written in modern C++ and only has the standard library and OpenMP as dependencies. The target platforms are Linux and OS X, with little work expected to adapt it to other platforms. The code was optimized for speed as well as readability (as we hope to give a reference implementation).

7.2 Decider Setting

In this section we test the running time performance of our new decider algorithm (Algorithm 3). We first describe our new benchmark using the three data sets, and then discuss our experimental findings, in particular how the performance and improvement over the state of the art varies with the distance and also the "neighbor rank" in the data set.

Benchmark. For the decider, we want to specifically test how the decision distance δ and how the choice of the second curve σ influences the running time of the decider. To experimentally evaluate this, we create a benchmark for each data set $\mathcal{D}$ in the following way. We select a random curve $\pi \in \mathcal{D}$ and sort the curves in the data set $\mathcal{D}$ by their distance to π in increasing order, obtaining the sequence $\sigma_1, \ldots, \sigma_n$. We define the *neighbor rank* of σ_i with respect to π to be its index i in the ordering. To create the benchmark, for all $k \in \{1, \ldots, \lfloor \log n \rfloor\}$ we

- select a curve $\sigma \in \{\sigma_{2^k}, \ldots, \sigma_{2^{k+1}-1}\}$ uniformly at random[3],

- compute the exact distance $\delta^* := d_F(\pi, \sigma)$,

- for each $l \in \{-10, \ldots, 0\}$, add benchmark tests $(\pi, \sigma, (1-2^l)\cdot\delta^*)$ and $(\pi, \sigma, (1+2^l)\cdot\delta^*)$.

By repeating this process for 1000 uniformly random curves $\pi \in \mathcal{D}$, we create 1000 test cases for every pair of k and l.

Running Times. First we show how our implementation performs in this benchmark. In Figure 11 we depict timings for running our implementation on the benchmark for all data sets. We can see that decision distances larger than the exact Fréchet distance are harder than smaller decision distances. This effect is most likely caused by the fact that decider instances with positive result need to find a path through the free-space diagram, while negative instances might be resolved earlier as it already becomes clear close to the lower left corner of the free-space diagram that there cannot exist such a path. Also, the performance of the decider is worse for computations on $(\pi, \sigma, c \cdot \delta^*)$ when σ has a small neighbor rank (with respect to π) and c is close to 1. This seems natural, as curves which are closer are more likely in the data set to actually be of similar shape, and similar shapes often lead to bottlenecks in the free-space diagram (i.e., small regions where a witness path can barely pass through), which have to be resolved in much more detail and therefore lead to a higher number of recursive calls. It follows that the benchmark instances for low k and l close to 0 are the hardest; this is the case for all data sets. In CHARACTERS we can also see that for $k = 7$ there is suddenly a rise in the running time for certain distance factors. We assume that this comes from the fact that the previous values of k all correspond to the same written character and this changes for $k = 7$.

We also run the original code of the winner of the GIS Cup, namely [6], on our benchmark and compare it with the running time of our implementation. See Figure 12 for the speed-up factors of our implementation over the GIS Cup winner implementation. The speed-ups obtained depend on the data set. While for every data set a significant amount of benchmarks for different k and l are more than one order of magnitude faster, for GEOLIFE even speed-ups by 2 orders of magnitude are reached. Speed-ups tend to be higher for larger distance factors. The results on GEOLIFE suggest that for longer curves, our implementation becomes significantly faster relative to the current state of the art. Note that there also are situations where our decider shows similar performance to the one of [6]; however, those are cases where both deciders can easily recognize that the curves are far (due to, e.g., their start or end points being far). We additionally show the percentage of instances that are already decided by the filters in Figure 13.

7.3 Influence of the Individual Pruning Rules

We also verified that the improvements that we introduced indeed are all necessary. In Section 3.3 we introduced six pruning rules. Rule I, i.e., "Empty Inputs", is essential. If we

---

[3]Note that for $k = \lfloor \log n \rfloor$ some curves might be undefined as possibly $2^{k+1} - 1 > n$. In this case we select a curve uniformly at random from $\{\sigma_{2^k}, \ldots, \sigma_n\}$.

SIGSPATIAL:

CHARACTERS:

GEOLIFE:
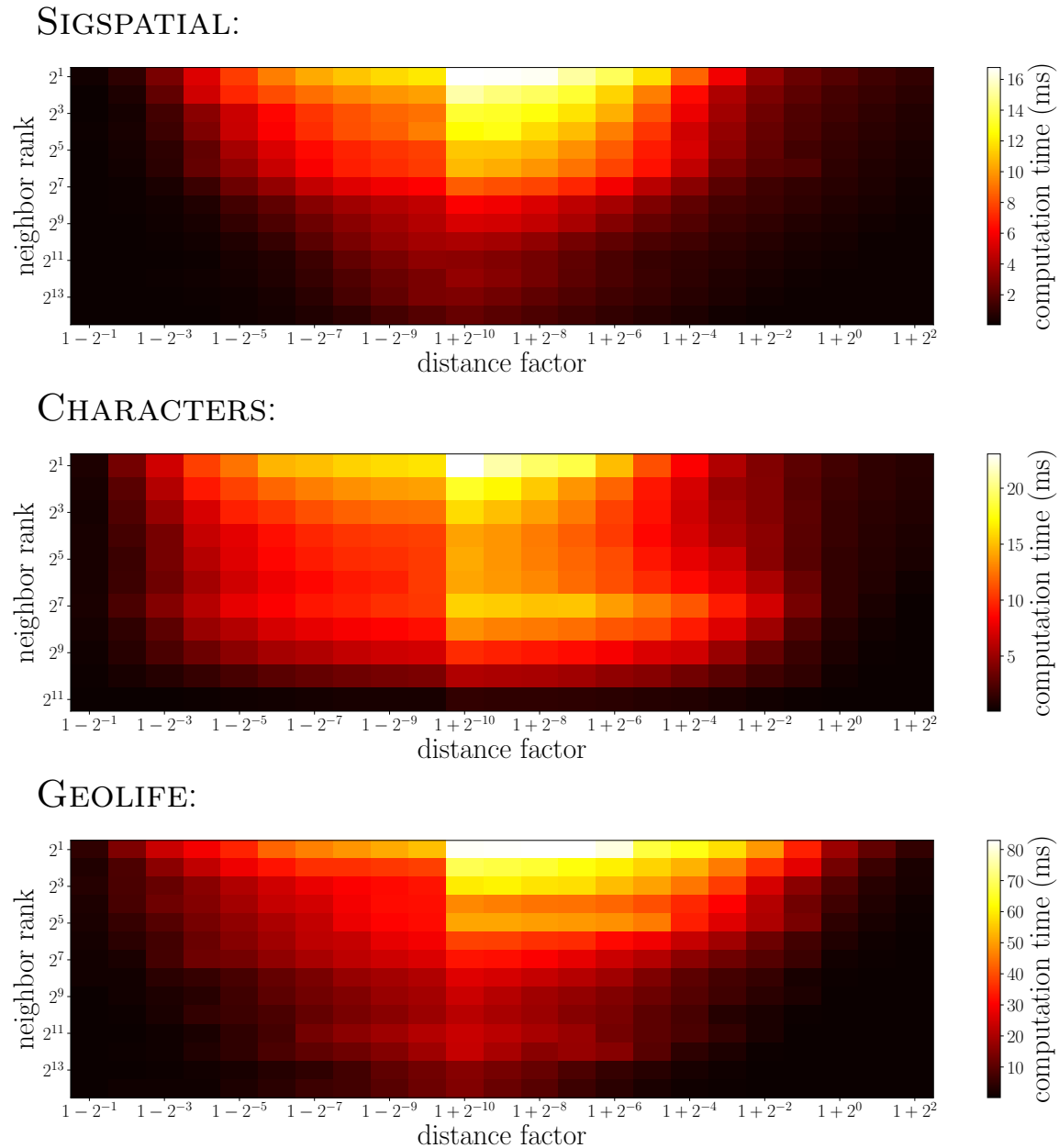
Figure 11: Running times of the decider benchmark when we run our implementation on it.

SIGSPATIAL:

CHARACTERS:

GEOLIFE:
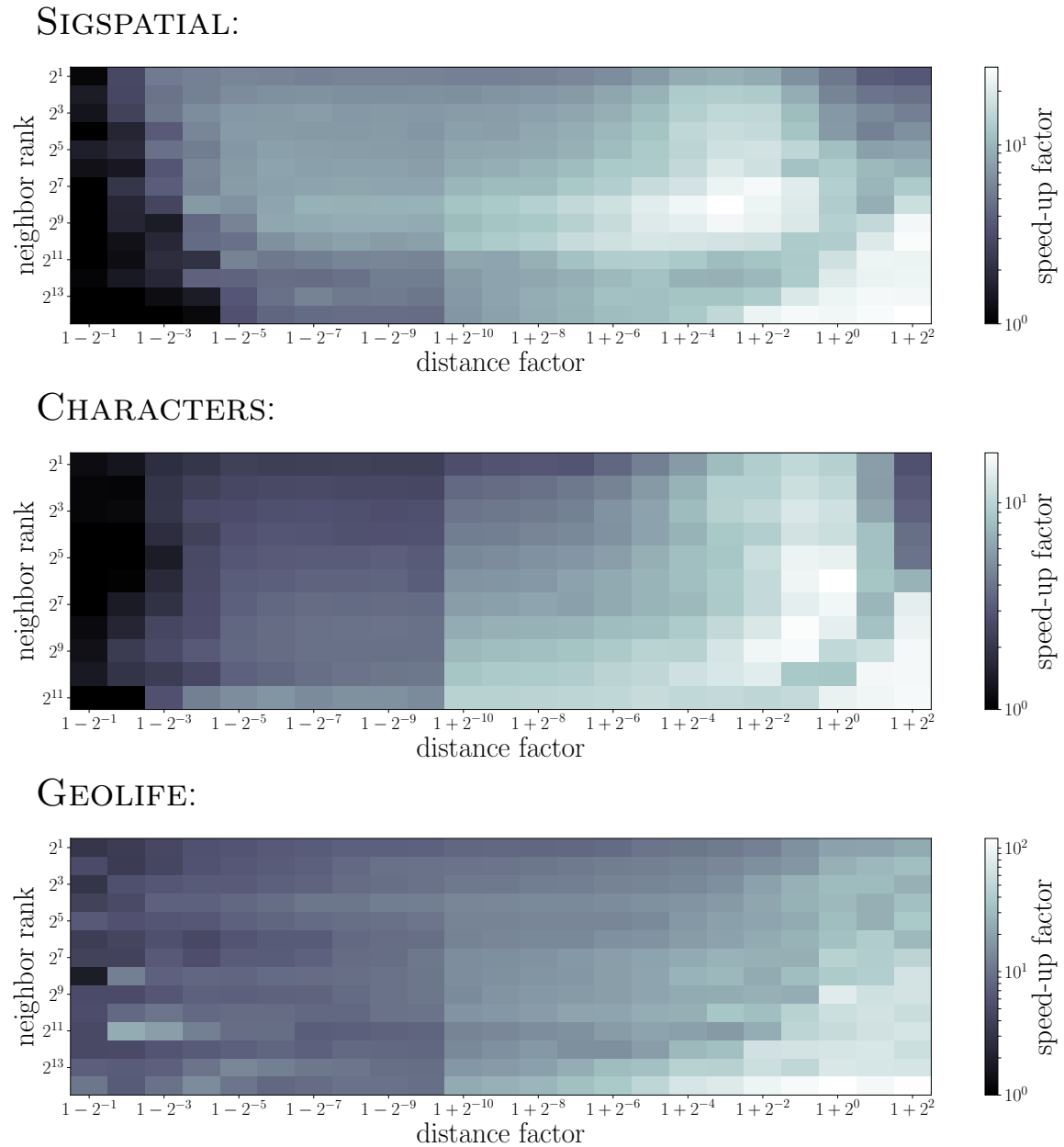
Figure 12: The speed-up factors obtained over the GIS Cup winner on the decider benchmark.

Sᴵɢsᴘᴀᴛɪᴀʟ:

Cʜᴀʀᴀᴄᴛᴇʀs:

Gᴇᴏʟɪꜰᴇ:
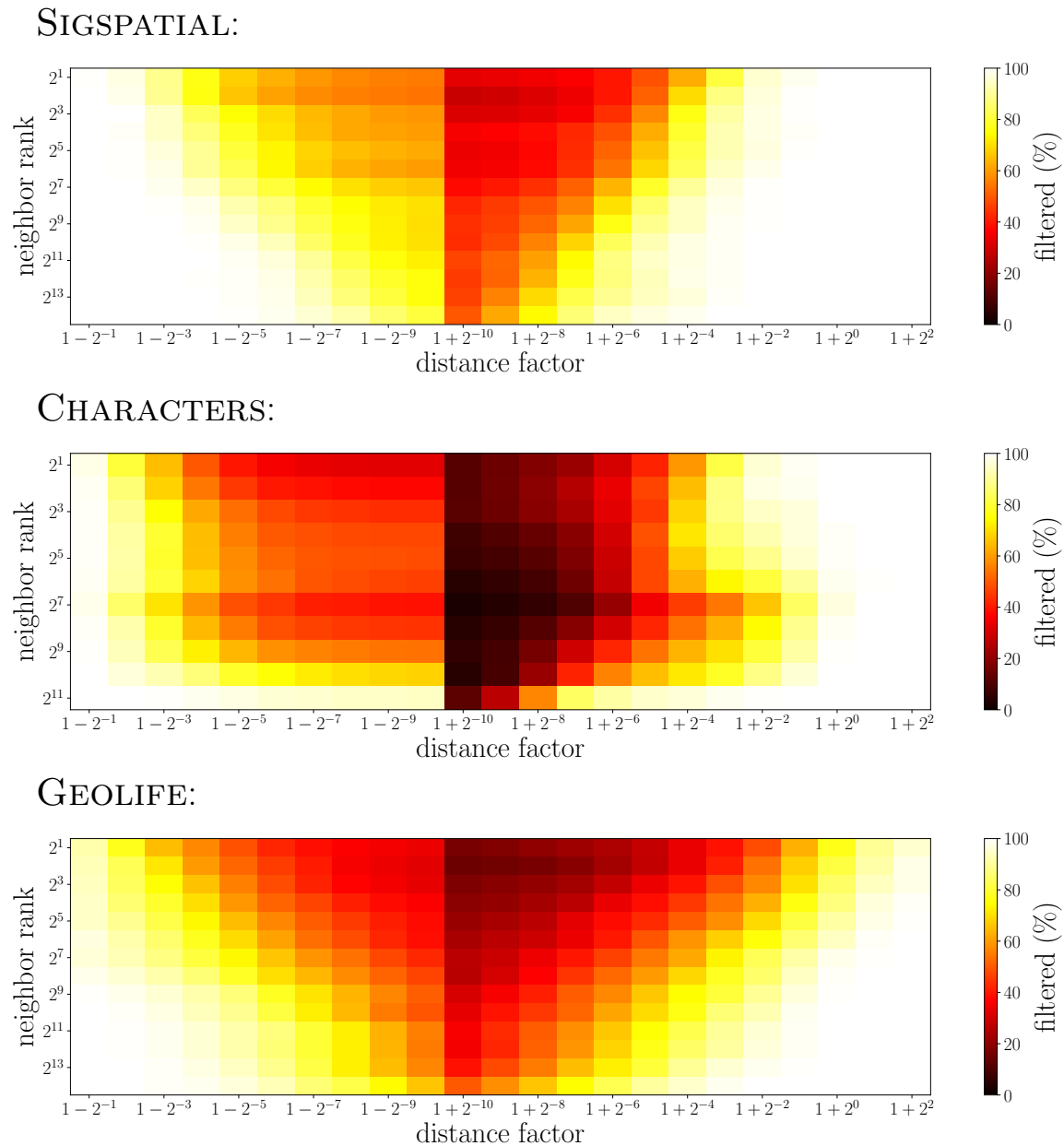
Figure 13: The percentage of queries that are decided by the filters on the decider benchmark.

Table 2: Times (in ms) for running the decider benchmarks with leaving out pruning steps. We only ran the first 100 queries for each k and l due to large running times when omitting the third rule.

	SIGSPATIAL	CHARACTERS	GEOLIFE
omit none	99.085	153.195	552.661
omit Rule II	112.769	204.347	1382.306
omit Rule IIIa	193.437	296.679	1779.810
omit Rule IIIb	5317.665	1627.817	385031.421
omit Rule IIIc	202.469	273.146	2049.632
omit Rule IV	110.968	161.142	696.382

were to omit it, we would hardly improve over the naive free-space exploration algorithm. The remaining five rules can potentially be omitted. Thus, for each of these pruning rules, we let our implementation run on the decider benchmark with this single rule disabled; and once with all rules enabled. See Table 2 for the results. Clearly, all pruning rules yield significant improvements when considering the timings of the GEOLIFE benchmark. All rules, except Rule IV, also show significant speed-ups for the other two data sets. Additionally, note that omitting Rule IIIb drastically increases the running time. This effect results from Rule IIIb being the main rule to prune large reachable parts, which we otherwise have to explore completely. One can clearly observe this effect in Figure 9.

Filters.   In Figure 13 we show what percentage of the queries are decided by the filters. We can see that the closer we get to the actually distance δ^* of two curves, the less likely it gets that the filters can make a decision. Furthermore, for the distances that are greater than δ^* the filters perform worse than for distances less than δ^*. We additionally observe that on CHARACTERS the filters perform significantly worse than on the other two data sets. Also the running times are inversely correlated with the percentage of decisions of the filters as returning earlier in the decider naturally reduces the overall runtime.

7.4   Query Setting

We now turn to the experiments conducted for our query data structure, which we explained in Section 5.

Benchmark.   We build a query benchmark similar to the one used in [6]. For each $k \in \{0, 1, 10, 100, 1000\}$, we select a random curve $\pi \in \mathcal{D}$ and then pick a threshold distance δ such that a query of the form (π, δ) returns exactly $k+1$ curves (note that the curve π itself is also always returned). We repeat this 1000 times for each value of k and also create such a benchmark for each of the three data sets.

Running Times.   We compare our implementation with the top three implementations of the GIS Cup on this benchmark. The results are shown in Table 3. Again the running
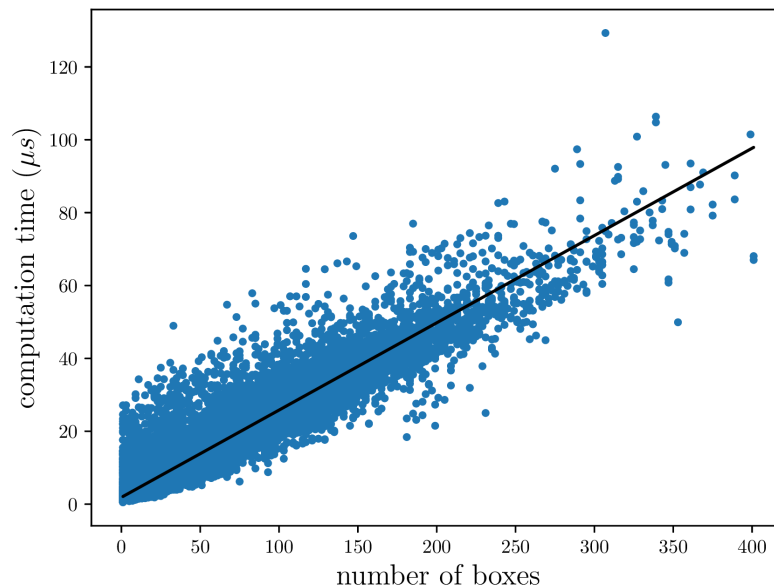
Figure 14: Shows how much time a call to the complete decider takes plotted over the number of boxes that the free-space diagram creates in total (i.e., even if a box is later split, it is still counted). The data are all exact computations (i.e., those where neither kd-tree nor filter decided) issued for the SIGSPATIAL query benchmark. The black line is the linear regression ($r^2 = 0.91$).

time improvement of our implementation depends on the data set. For CHARACTERS the maximal improvement factor over the second best implementation is 14.6, for SIGSPATIAL 17.3, and for GEOLIFE 29.1. For SIGSPATIAL and CHARACTERS it is attained at $k = 1000$, while for GEOLIFE it is reached at $k = 100$ but $k = 1000$ shows a very similar but slightly smaller factor.

To give deeper insights about the single parts of our decider, a detailed analysis of the running times of the single parts of the algorithm is shown in Table 4. Again we witness different behavior depending on the data set. It is remarkable that for SIGSPATIAL the running time for $k = 1000$ is dominated by the greedy filter. This suggests that improving the filters might still lead to a significant speed-up in this case. However, for most of the remaining cases the running time is clearly dominated by the complete decider, suggesting that our efforts of improving the state of the art focused on the right part of the algorithm.

7.5   Other Experiments

The main goal of the complete decider was to reduce the number of recursive calls that we need to consider during the computation of the free-space diagram. Due to our optimized algorithm to compute simple boundaries with adaptive step size, we expect roughly a constant (or possibly polylogarithmic) running time effort per box, essentially independent of the size of the box. To test this hypothesis, we ask whether the number of recursive calls

Table 3: Comparing the running times (in s) of the queries of the top three implementations of the GIS Cup 2017 with our new implementation on the query benchmark on all data sets (1000 queries per entry).

k	SIGSPATIAL					CHARACTERS					GEOLIFE				
	0	1	10	100	1000	0	1	10	100	1000	0	1	10	100	1000
[6]	0.094	0.123	0.322	1.812	8.408	0.187	0.217	0.421	2.222	17.169	0.298	0.741	4.327	33.034	109.44
[12]	0.421	0.618	1.711	7.86	35.704	0.176	0.28	0.611	3.039	17.681	3.627	6.067	26.343	120.509	415.548
[22]	0.197	0.188	0.643	5.564	76.144	0.142	0.147	0.222	1.849	22.499	2.614	4.112	16.428	166.206	1352.19
ours	0.017	0.007	0.026	0.130	0.490	0.004	0.020	0.058	0.301	1.176	0.027	0.089	0.341	1.108	3.642

Table 4: Timings (in s) of the single parts of our query algorithm on the query benchmark on all three data sets. To avoid confusion, note that the sum of the times in this table do not match the entries in Table 3 as those are parallelized timings and additionally the timing itself introduces some overhead.

k	SIGSPATIAL					CHARACTERS					GEOLIFE				
	0	1	10	100	1000	0	1	10	100	1000	0	1	10	100	1000
spatial hashing	0.002	0.003	0.005	0.017	0.074	0.002	0.002	0.004	0.011	0.032	0.006	0.009	0.016	0.032	0.091
greedy filter	0.004	0.006	0.024	0.143	0.903	0.004	0.010	0.032	0.153	0.721	0.009	0.017	0.060	0.273	1.410
equal-time filter	0.000	0.001	0.006	0.030	0.088	0.001	0.004	0.018	0.088	0.424	0.005	0.017	0.063	0.273	1.211
negative filter	0.001	0.002	0.010	0.044	0.107	0.003	0.012	0.038	0.152	0.309	0.008	0.020	0.069	0.200	0.606
complete decider	0.002	0.011	0.044	0.214	0.330	0.005	0.030	0.109	0.671	2.639	0.062	0.210	0.998	3.025	8.760

is indeed correlated with the running time. To test this, we measured the time for each complete decider call in the query benchmark and plotted it over the number of boxes that were considered in this call. The result of this experiment is shown in Figure 14. We can see a practically (near-)linear correlation between the number of boxes and the running time.

8 Certificates

Whenever we replace a naive implementation in favor of a fast, optimized, but typically more complex implementation, it is almost unavoidable to introduce bugs to the code. As a useful countermeasure the concept of *certifying algorithms* has been introduced; we refer to [27] for a survey. In a nutshell, we aim for an implementation that outputs, apart from the desired result, also a proof of correctness of the result. Its essential property is that the certificate should be *simple* to check (i.e., much simpler than solving the original problem). In this way, the certificate gives any user of the implementation a simple means to check the output for any conceivable instance.

Following this philosophy, we have made our implementation of the Fréchet decider *certifying*: for any input curves π, σ and query distance δ, we are able to return, apart from the output whether the Fréchet distance of π and σ is at most δ, also a certificate c. On our realistic benchmarks, constructing this certificate slows down the Fréchet decider by roughly 50%. The certificate c can be checked by a simple verification procedure consisting of roughly 200 lines of code.

In Sections 8.1 and 8.2, we define our notion of YES and NO certificates, prove that they indeed certify YES and NO instances and discuss how our implementation finds them. In Section 8.3, we describe the simple checking procedure for our certificates. Finally, we conclude with an experimental evaluation in Section 8.4.

8.1 Certificate for YES Instances

To verify that $d_F(\pi, \sigma) \leq \delta$, by definition it suffices to give a feasible traversal, i.e., monotone and continuous functions $f : [0,1] \to [1,n]$ and $g : [0,1] \to [1,m]$ such that for all $t \in [0,1]$, we have $(\pi_{f(t)}, \sigma_{g(t)}) \in F$, where $F = \{(p,q) \in [1,n] \times [1,m] \mid \|\pi_p - \sigma_q\| \leq \delta\}$ denotes the free-space (see Section 3.1). We slightly simplify this condition by discretizing $(f(t), g(t))_{t \in [0,1]}$, as follows.

Definition 1. We call $T = (t_1, \ldots, t_\ell)$ with $t_i \in [1,n] \times [1,m]$ a YES certificate if it satisfies the following conditions: (See also Figure 15 for an example.)

1. (start) $t_1 = (1,1) \in F$,

2. (end) $t_\ell = (n,m) \in F$,

3. (step) For any $t_k = (p,q)$ and $t_{k+1} = (p', q')$, we have either

   (a) $p' = p$ and $q' > q$: Here, we require that $(p, \bar{q}) \in F$ for all $\bar{q} \in \{q, \lceil q \rceil, \ldots, \lfloor q' \rfloor, q'\}$,
   (b) $q' = q$ and $p' > p$: Here, we require that $(\bar{p}, q) \in F$ for all $\bar{p} \in \{p, \lceil p \rceil, \ldots, \lfloor p' \rfloor, p'\}$,
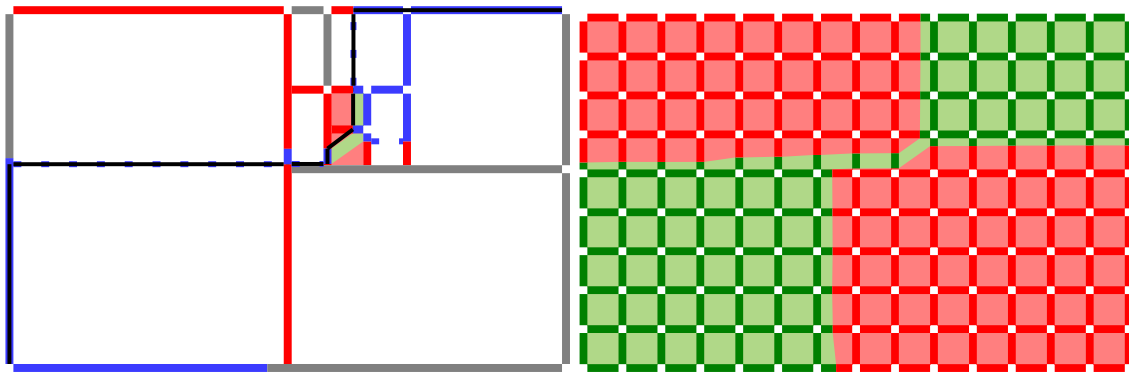
Figure 15: Example of a YES instance and its certificate. The right picture shows the free-space of the instance. The left picture illustrates the parts of the free-space explored by our algorithm and indicates the computed YES certificate by black lines.

(c) $i \leq p < p' \leq i+1$, $j \leq q < q' \leq j+1$ for some $i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}$: Here, we require that $(p,q), (p',q') \in F$.

It is straightforward to show that a YES certificate T proves correctness for YES instances as follows.

Proposition 1. *Any YES certificate $T = (t_1, \ldots, t_\ell)$ with $t_i \in [1,n] \times [1,m]$ proves that $d_F(\pi, \sigma) \leq \delta$.*

Proof. View T as a polygonal curve in $[1,n] \times [1,m]$ and let $\tau : [0,1] \to [1,n] \times [1,m]$ be a reparameterization of T. Let f, g be the projection of τ to the first and second coordinate, respectively. Note that by the assumption on T, f and g are monotone and satisfy $(f(0), g(0)) = (1,1)$ and $(f(1), g(1)) = (n,m)$. We claim that $(f(t), g(t)) \in F$ for all $t \in [0,1]$, which thus yields $d_F(\pi, \sigma) \leq \delta$ by definition.

To see the claim, we recall that for any *cell* $[i, i+1] \times [j, j+1]$, the free-space restricted to this cell, i.e., $F \cap [i, i+1] \times [j, j+1]$, is convex (as it is the intersection of an ellipse with $[i, i+1] \times [j, j+1]$, see [2]). Observe that for any segment from $t_k = (p,q)$ to $t_{k+1} = (p', q')$, we (implicitly) decompose it into subsegments contained in single cells (e.g., for $p' = p$ and $q' > q$, the segment from (p,q) to (p,q') is decomposed into the segments connecting the sequence $(p,q), (\lceil p \rceil, q), \ldots, (\lfloor p' \rfloor, q), (p'q')$. As each such subsegment is contained in a single cell, by convexity we see that the whole subsegment is contained in F if the corresponding endpoints of the subsegment are in F. This concludes the proof. □

It is not hard to prove that for YES instances, such a certificate always exists (in fact, there always is a certificate of length $O(n + m)$). Furthermore, for each YES instance in our benchmark set, our implementation indeed finds and returns a YES instance, in a way we describe next.

Certifying positive filters. It is straightforward to construct YES certificates for instances that are resolved by our positive filters (Bounding Box Check, Greedy and Adaptive Equal-

Time): All of these filters implicitly construct a feasible traversal. In particular, for any instance for which the Bounding Box Check applies (which shows that any pair of points of π and σ are within distance δ), already the sequence $((1,1),(n,1),(n,m))$ yields a YES certificate.

For Greedy, note that the sequence of positions (i,j) visited in Algorithm 4 yields a YES certificate: Indeed, any step from (i,j) is either a vertical step to $(i,j+s)$ (corresponding to case 3a), a horizontal step to $(i+s,j)$ (corresponding to case 3b), or a diagonal step *within a cell* to $(i+1,j+1)$ (corresponding to Case 3c of Definition 1). Furthermore, such a step is only performed if it stays within the free-space.

Finally, for Adaptive Equal-Time, we also record the sequence of positions (i,j) visited in Algorithm 4 (recall that here, we change the set of possible steps for $s > 1$ to $S = \{(i+s,j+s')\}$ with $s' = \lfloor \frac{m-j}{n-i} \cdot s \rfloor$) – with the only difference that we need to replace any step from (i,j) to $(i+s,j+s')$ by the sequence $(i,j),(i+s,j),(i+s,j+s')$. Note that this sequence satisfies Condition (step) of Definition 1, as Adaptive Equal-Time only performs this step if it can verify that *all pairwise distances* between $\pi_{i...i+s}$ and $\sigma_{j...j+s'}$ are bounded by δ.

Certifying YES instances in the complete decider. Recall that the complete decider via free-space exploration decides an instance by recursively determining, given the inputs B_l^R, B_b^R of a box B, the corresponding outputs B_r^R, B_t^R. In particular, YES instances are those with $(n,m) \in B_t^R$ (or equivalently $(n,m) \in B_r^R$) for the box $B = [1,n] \times [1,m]$. To certify such instances, we memorize for each point in B_r^R and B_t^R a predecessor of a feasible traversal from $(1,1)$ to this point. Note that here, it suffices to memorize such a predecessor only for the *first*, i.e., lowest or leftmost, *point of each interval* in B_r^R and B_t^R (as any point in this interval can be reached by traversing to the first point of the interval and then along this reachable interval to the destination point). This gives rise to a straightforward recursive approach to determine a feasible traversal.

In the complete decider, whenever we determine some output B_t^R, it is because of one of the following reasons: (1) one of of our pruning rules is successful, (2) the box B is on the cell-level, or (3) we determine B_t^R as the union of the outputs $(B_1)_t^R, (B_2)_t^R$ of the boxes B_1, B_2 obtained by splitting B vertically. Note that we only need to consider the case in which B_t^R is determined as non-empty (otherwise nothing needs to be memorized). Let us consider each case separately.

If reason (1) determines a non-empty B_t^R, then this happens either by Rule IIIb or by Rule IIIc. Note that in both cases, B_t^R consists of a single interval. If Rule IIIb applies, then the last, i.e., topmost, point on B_l is reachable and proves that the *free prefix* of B_t^R is reachable. Thus, we store *the last interval of B_l^R* as the responsible interval for the (single) interval in B_t^R. Similarly, if Rule IIIc applies, then consider the first, i.e., leftmost, point $(i, j_{\max})$ on B_t. Since the rule applies, the opposite point $(i, j_{\min})$ on B_l must be reachable and the path $\{i\} \times [j_{\min}, j_{\max}]$ must be free. Thus, we can store *the interval of B_b^R containing* $(i, j_{\min})$ as the responsible interval for the (single) interval in B_t^R.

If reason (2) determines a non-empty B_t^R, then we are on a cell-level. In this case, either B_l^R or B_b^R is a non-empty interval, and we can store such an interval as the responsible

interval for the (single) interval in B_t^R. Finally, if reason (3) determines a non-empty B_t^R, then we simply keep track of the responsible interval for each interval in $(B_1)_t^R$ and $(B_2)_t^R$ (to be precise, if the last interval of $(B_1)_t^R$ and the first interval of $(B_2)_t^R$ overlap by the boundary point, we merge the two corresponding intervals and only keep track of the responsible interval of the last interval of $(B_1)_t^R$ and can safely forget about the responsible interval of the first interval of $(B_2)_t^R$.

Note that we proceed analogously for outputs B_r^R. Furthermore, the required memorization overhead is very limited.

It is straightforward to use the memorized information to compute a YES certificate recursively: Specifically, to compute a YES certificate reaching some point x on an output interval I, we perform the following steps. Let J be the responsible interval of I. We recursively determine a YES certificate reaching the first point J. Then we append a point to the certificate to traverse to the point of J from which we can reach the first point of I (this point is easily determined by distinguishing whether we are on the cell-level, and whether J is opposite to I or intersects I in a corner point). We append the first point of I to the certificate, and finally append the point x to the certificate.[4] By construction, the corresponding traversal never leaves the free-space. Using this procedure, we can compute a YES certificate by computing a YES certificate reaching (n, m) on the last interval of B_t^R for the initial box $B = [1, n] \times [1, m]$.

8.2   Certificate for NO Instances

We say that a point (p, q) lies on the bottom boundary if $q = 1$, on the right boundary if $p = n$, on the top boundary if $q = m$, and on the left boundary if $p = 1$. Likewise, we say that a point (p', q') lies to the lower right of a point (p, q), if $p \leq p'$ and $q \geq q'$.

Definition 2. We call $T = (t_1, \ldots, t_\ell)$ with $t_i \in [1, n] \times [1, m]$ a NO certificate if it satisfies the following conditions: (See also Figure 16 for an example.)

1. (start) t_1 lies on the right or bottom boundary and $t_1 \notin F$,

2. (end) t_ℓ lies on the left or upper boundary and $t_\ell \notin F$,

3. (step) For any $t_k = (p, q)$ and $t_{k+1} = (p', q')$, we have either

   (a) $p' = p$ and $q' > q$: Here, for any neighboring elements $\bar{q}_1, \bar{q}_2$ in $q, \lceil q \rceil, \ldots, \lfloor q' \rfloor, q'$, we require that $(\{p\} \times [\bar{q}_1, \bar{q}_2]) \cap F = \emptyset$,

   (b) $q' = q$ and $p' < p$: Here, for any neighboring elements $\bar{p}_1, \bar{p}_2$ in $p', \lceil p' \rceil, \ldots, \lfloor p \rfloor, p$, we require that $([\bar{p}_1, \bar{p}_2] \times \{q\}) \cap F = \emptyset$,

   (c) t_{k+1} lies to the lower right of t_k, i.e., $p \leq p'$ and $q \geq q'$.

We prove that a NO certificate T proofs correctness for NO instances as follows.

Proposition 2. *Any NO certificate $T = (t_1, \ldots, t_\ell)$ with $t_i \in [1, n] \times [1, m]$ proves that $d_F(\pi, \sigma) > \delta$.*

---

[4]To be precise, we only append a point if it is different from the last point of the current certificate.
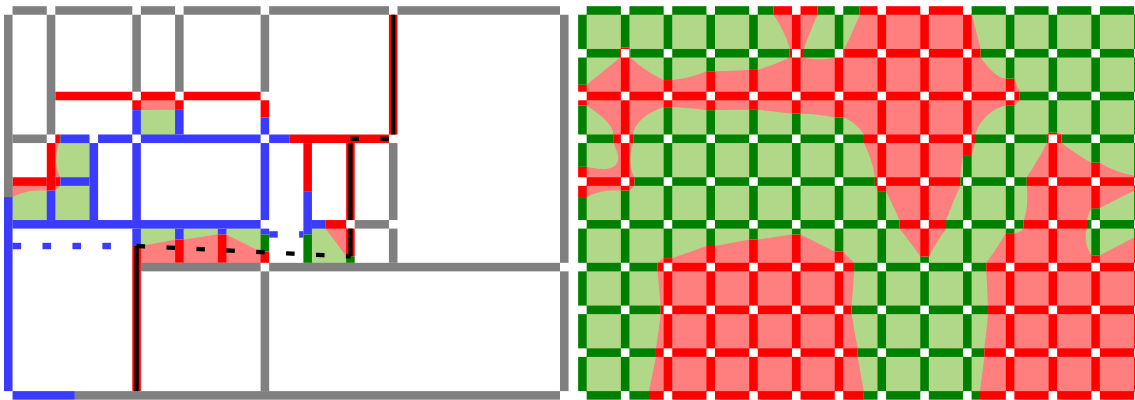
Figure 16: Example of a NO instance and its certificate. The right picture shows the free-space of the instance. The left picture illustrates the parts of the free-space explored by our algorithm and indicates the computed NO certificate by black lines.

Proof. We inductively prove that no feasible traversal from $(1,1)$ to (n,m) can visit any point to the lower right of t_i, for all $1 \leq i \leq \ell$. As an immediate consequence, $d_F(\pi, \sigma) > \delta$, since t_ℓ lies on the left or upper boundary and thus *any* feasible traversal must visit a point to the lower right of t_ℓ – hence, such a traversal cannot exists.

As base case, note that t_1 lies on the right or bottom boundary and is not contained in the free-space. Thus, by monotonicity, no feasible traversal can visit any point to the lower right of t_1. Thus, assume that the claim is true for $t_i = (p, q)$ and consider the next point $t_{i+1} = (p', q')$ in the sequence. If t_{i+1} lies to the lower right of t_i, the claim is trivially fulfilled for t_{i+1} by monotonicity. If, however, $p' = p$ and $q' > q$, then Condition 3a of Definition 2 is equivalent to $(p \times [q, q']) \cap F = \emptyset$. Note that any feasible traversal visiting a point to the lower right of t_{i+1} must either visit a point to the lower right of t_i – which is not possible by inductive assumption – or must cross the path $\{p\} \times [q, q']$ – which is not possible as $(\{p\} \times [q, q']) \cap F = \emptyset$. We argue symmetrically for the case that $q' = q$ and $p' < p$. This concludes the proof. $\qquad\square$

Note that our definition of NO certificate essentially coincides with the definition of a cut of positive width in [14]. For NO instances, such a NO certificate always exists (in contrast to YES certificates, the shortest such certificate is of length $\Theta(n^2)$ in the worst case). For all NO instances in our benchmark sets, our implementation manages to find and return such a NO certificate, in a way we describe next.

Certifying the negative filter. It is straightforward to compute a NO certificate for instances resolved by our negative filter. Note that this filter, if successful, determines an index $p \in \{1, \ldots, n\}$ such that π_p is far from all points on σ, or symmetrically an index $q \in \{1, \ldots, m\}$ such that σ_q is far from all points on π. Thus, in these cases, we can simply return the NO certificate $((p, 1), (p, m))$ or $((n, q), (1, q))$, respectively.

Algorithm 7 High-level code for computing a NO certificate.

1: **procedure** COMPUTENOCERTIFICATE(π, σ, δ)
2:     $N \leftarrow$ non-free segments determined by COMPLETEDECIDER(π, σ, δ)
3:     $Q \leftarrow \{I \in N \mid \text{lowerRight}(I)$ lies on bottom or right boundary$\}$
4:     Build orthogonal range search data structure D,
5:                                     storing all $I \in N \setminus Q$ under the key lowerRight(I).
6:     **while** $Q \neq \emptyset$ **do**
7:         Pop any element I from Q
8:         **if** upperLeft(I) lies on top or left boundary **then**
9:             Reconstruct sequence of intervals leading to I
10:             **return** corresponding NO certificate
11:         **else**
12:             $Q' \leftarrow D.$REPORTANDDELETE(upperLeft(I))
13:                     $\triangleright$ reports J if lowerRight(J) is to the lower right of upperLeft(I)
14:             $Q \leftarrow Q \cup Q'$

Certifying NO instances in the complete decider. Whenever the complete decider via free-space exploration returns a negative answer, the explored parts of the free-space diagram must be sufficient to derive a negative answer. This gives rise to the following approach: Consider all non-free segments computed by the complete decider. We start from a non-free segment touching the bottom or right boundary and traverse non-free segments (possibly also making use of monotonicity steps according to Case 3c of Definition 2) and stop as soon as we have found a non-free segment touching the left or top boundary.

Formally, consider Algorithm 7. Here, we use the notation that lowerRight(I) denotes the lower right endpoint of I, i.e., the right endpoint if I is a horizontal segment and the lower endpoint if I is a vertical segment. Analogously, upperLeft(I) denotes the upper left endpoint of I.

The initial set of non-free segments in Algorithm 7 consists of the non-free segments of *all simple boundaries* determined by the complete decider via free-space exploration. We maintain a queue Q of non-free segments, which initially contains all non-free segments touching the right or bottom boundary. Furthermore, we maintain a data structure D of yet *unreached* non-free intervals. Specifically, we require D to store intervals I under the corresponding key lowerRight(I) $\in [1, n] \times [1, m]$ in a way to support the query REPORTANDDELETE(p): Such a query returns all $I \in D$ such that lowerRight(I) lies to the lower right of p and deletes all returned intervals from D.

Equipped with such a data structure, we can traverse all elements in the queue as follows: We delete any interval I from Q and check whether it reaches the upper or left boundary. If this is the case, we have (implicitly) found a NO certificate, which we then reconstruct (by memorizing why each element of the queue was put into the queue). Otherwise, we add to Q all intervals from D that can be reached by a monotone step (according to Case 3c of Definition 2) from upperLeft(I); these intervals are additionally deleted from D.

To implement D, we observe that it essentially asks for a 2-dimensional orthogonal range search data structure where the ranges are unbounded in two directions (and bounded in the other two). Already for the case of 2-dimensional ranges with only a single unbounded direction (sometimes called 1.5-dimensional), a very efficient solution is provided by a classic data structure due to McCreight, the priority search tree data structure [28]. We can adapt it in a straightforward manner to implement D such that it (1) takes time $O(d \log d)$ and space $O(d)$ to construct D on an initial set of size d and (2) supports REPORTANDDELETE(p) queries in time $O(k + \log d)$, where k denotes the number of reported elements. Thus, Algorithm 7 can be implemented to run in time $O(|N| \log |N|)$.

8.3 Certificate Checker

It remains to describe how to check the correctness of a given certificate $T = (t_1, \ldots, t_\ell)$. For this, we simply verify that all properties of Definition 1 or Definition 2 are satisfied.

Checking YES certificates. Observe that the only conditions in the definition of YES instances are either simple comparisons of neighboring elements t_k, t_{k+1} in the sequence or *freeness tests*, specifically, whether a give position $p \in [1, n] \times [1, m]$ is free, i.e, whether π_{p_1} and σ_{p_2} have distance at most δ. The latter test only requires interpolation along a curve segment (to obtain π_{p_1} and σ_{p_2}) and a Euclidean distance computation. Thus, YES certificates are extremely simple to check.

Checking NO certificates. Checking NO certificates involves a slightly more complicated geometric primitive than the freeness tests of YES certificates. Apart from simple comparisons of neighboring elements t_k, t_{k+1}, the conditions in the definition involve the following *non-freeness tests*: Given a (sub)segment $\pi_{p..p'}$ with $i \leq p \leq p' \leq i + 1$ for some $i \in [n]$, as well as a point σ_q with $q \in [1, m]$, determine whether all points on $\pi_{p..p'}$ have distance strictly larger than δ from σ_q. Besides the (simple) interpolation along a line segment to obtain σ_q, we need to determine intersection points of the line containing $\pi_{p..p'}$ and the circle of radius δ around σ_q (if these exist). From these intersection points, we verify that $\pi_{p..p'}$ and the circle do not intersect, concluding the check.

In summary, certificate checkers are straightforward and simple to implement.

8.4 Certification Experiments

We evaluate the overhead introduced by computing certificates using our benchmark sets for the query setting. In particular, as our implementation can be compiled both as a certifying and a non-certifying version, we compare the running times of both versions. The results are depicted in Table 5. Notably, the slowdown factor introduced by computing certificates ranges between 1.29 and 1.46 (SIGSPATIAL), 1.44 and 1.67 (CHARACTERS) and 1.42 and 1.73 (GEOLIFE). As expected, the certificate computation time is dominated by the task of

Table 5: Certificate computation and check times on query setting benchmark (in ms). The first and second bold lines show the running time of our implementation compiled without and with certification, respectively. For the certifying variant, we also give the times to compute YES and NO certificates of the complete decider (note that filter certificates are computed on the fly by the filters and hence cannot be separately measured; also, this certificate computation time does not include the overhead to record additional information during the complete decision procedure). Finally, we give running times for checking correctness of certificates.

	SIGSPATIAL				
k	0	1	10	100	1000
computation without certification	**6.9**	**21.3**	**84.5**	**429.4**	**1409.1**
certifying computation	**10.0**	**29.6**	**117.8**	**553.8**	**1840.2**
–computation of certificates	1.0	3.7	12.0	40.9	65.7
–YES certificates (complete decider)	0.0	0.4	1.6	8.2	12.1
–NO certificates (complete decider)	1.0	3.3	10.0	31.4	50.0
checking certificates	**6.4**	**13.9**	**63.7**	**426.5**	**3803.2**
–checking filter certificates	6.0	11.3	51.6	361.2	3666.7
–checking complete decider certificates	0.4	2.6	12.1	65.3	136.5

	CHARACTERS				
k	0	1	10	100	1000
computation without certification	**12.1**	**55.5**	**205.8**	**1052.8**	**4080.3**
certifying computation	**20.3**	**91.6**	**311.2**	**1589.8**	**5895.8**
–computation of certificates	4.0	20.0	59.6	220.7	470.2
–YES certificates (complete decider)	0.0	0.5	2.3	24.9	181.3
–NO certificates (complete decider)	3.9	19.2	56.3	186.2	259.4
checking certificates	**6.3**	**21.6**	**76.8**	**457.7**	**2626.1**
–checking filter certificates	5.3	14.8	49.7	278.0	1759.8
–checking complete decider certificates	1.0	6.8	27.2	179.6	866.3

	GEOLIFE				
k	0	1	10	100	1000
computation without certification	**82.2**	**251.1**	**1156.6**	**3663.1**	**11452.4**
certifying computation	**142.1**	**414.6**	**1834.4**	**5304.2**	**16248.7**
–computation of certificates	40.1	100.7	388.0	767.7	1827.8
–YES certificates (complete decider)	0.0	3.2	20.0	87.6	247.5
–NO certificates (complete decider)	39.7	96.7	364.5	664.8	1517.1
checking certificates	**70.9**	**185.2**	**733.7**	**3595.4**	**20188.4**
–checking filter certificates	45.2	85.9	283.8	1754.0	12987.5
–checking complete decider certificates	25.7	99.4	450.0	1841.4	7200.9

generating NO certificates (which is more complex than computing YES certificates), even for large values of k for which most unfiltered instances are YES instances.

At first sight, it might be surprising that checking the certificates takes longer than computing them. However, this is due to the fact that our filters often display sublinear running time behavior (by using the heuristic checks and adaptive step sizes). However, to keep our certificate checker elementary, we have not introduced any such improvements to the checker, which thus has to traverse essentially all points on the curves. This effect is particularly prominent for large values of k.

9   Conclusion

In this work we presented an implementation for computing the Fréchet distance which beats the state-of-the-art by one to two orders of magnitude in running time in the query as well as the decider setting. Furthermore, it can be used to compute certificates of correctness with little overhead. To facilitate future research, we created two benchmarks on several data sets – one for each setting – such that comparisons can easily be conducted. Given the variety of applications of the Fréchet distance, we believe that this result will also be of broader interest and implies significant speed-ups for other computational problems in practice.

This enables a wide range of future work. An obvious direction to continue research is to take it back to theory and show that our pruning approach provably has subquadratic runtime on a natural class of realistic curves. On the other hand, one could try to find further pruning rules or replace the divide-and-conquer approach by some more sophisticated search. To make full use of the work presented here, it would make sense to incorporate this algorithm in software libraries. Currently, we are not aware of any library with a non-naive implementation of a Fréchet distance decider or query. Finally, another possible research direction would be to work on efficient implementations for similar problems like the Fréchet distance under translation, rotation or variants of map matching with respect to the Fréchet distance. In summary, this paper should lay ground to a variety of improvements for practical aspects of curve similarity.

References

[1] Pankaj K. Agarwal, Rinat Ben Avraham, Haim Kaplan, and Micha Sharir. Computing the discrete Fréchet distance in subquadratic time. *SIAM J. Comput.*, 43(2):429–449, 2014.

[2] Helmut Alt and Michael Godau. Computing the Fréchet distance between two polygonal curves. *International Journal of Computational Geometry & Applications*, 5(01n02):75–91, 1995.

[3] Helmut Alt, Christian Knauer, and Carola Wenk. Comparison of distance measures for planar curves. *Algorithmica*, 38(1):45–58, 2004.

[4] Boris Aronov, Sariel Har-Peled, Christian Knauer, Yusu Wang, and Carola Wenk. Fréchet distance for curves, revisited. In Yossi Azar and Thomas Erlebach, editors, *Proc. 14th Annual European Symposium on Algorithms (ESA 2006)*, volume 4168 of *Lecture Notes in Computer Science*, pages 52–63. Springer, 2006.

[5] Maria Astefanoaei, Paul Cesaretti, Panagiota Katsikouli, Mayank Goswami, and Rik Sarkar. Multi-resolution sketches and locality sensitive hashing for fast trajectory processing. In *Proc. 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM GIS)*, 2018.

[6] Julian Baldus and Karl Bringmann. A fast implementation of near neighbors queries for Fréchet distance (GIS Cup). In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'17, pages 99:1–99:4, New York, NY, USA, 2017. ACM.

[7] Karl Bringmann. Why walking the dog takes time: Fréchet distance has no strongly subquadratic algorithms unless SETH fails. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 661–670. IEEE, 2014.

[8] Karl Bringmann and Marvin Künnemann. Improved approximation for Fréchet distance on c-packed curves matching conditional lower bounds. *Int. J. Comput. Geometry Appl.*, 27(1-2):85–120, 2017.

[9] Karl Bringmann and Wolfgang Mulzer. Approximability of the discrete Fréchet distance. *Journal of Computational Geometry*, 7(2):46–76, December 2015.

[10] Kevin Buchin, Maike Buchin, Wouter Meulemans, and Wolfgang Mulzer. Four soviets walk the dog: Improved bounds for computing the Fréchet distance. *Discrete & Computational Geometry*, 58(1):180–216, 2017.

[11] Kevin Buchin, Maike Buchin, and Yusu Wang. Exact algorithms for partial curve matching via the Fréchet distance. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 645–654. Society for Industrial and Applied Mathematics, 2009.

[12] Kevin Buchin, Yago Diez, Tom van Diggelen, and Wouter Meulemans. Efficient trajectory queries under the Fréchet distance (GIS Cup). In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'17, pages 101:1–101:4, New York, NY, USA, 2017. ACM.

[13] Kevin Buchin, Anne Driemel, Joachim Gudmundsson, Michael Horton, Irina Kostitsyna, and Maarten Löffler. Approximating (k, ℓ)-center clustering for curves. *CoRR*, abs/1805.01547, 2018.

[14] Kevin Buchin, Tim Ophelders, and Bettina Speckmann. SETH says: Weak Fréchet distance is faster, but only if it is continuous and in one dimension. In *SODA*, 2019. To appear.

[15] Jonathan Campbell, Jonathan Tremblay, and Clark Verbrugge. Clustering player paths. In *FDG*, 2015.

[16] Matteo Ceccarello, Anne Driemel, and Francesco Silvestri. FRESH: Fréchet similarity with hashing. *CoRR*, abs/1809.02350, 2018.

[17] Erin Chambers, Brittany Terese Fasy, Yusu Wang, and Carola Wenk. Map-matching using shortest paths. In *Proceedings of the 3rd International Workshop on Interactive and Spatial Computing*, pages 44–51. ACM, 2018.

[18] Character Trajectories Data Set. https://archive.ics.uci.edu/ml/datasets/Character+Trajectories. Accessed: 2018-12-03.

[19] Daniel Chen, Anne Driemel, Leonidas J Guibas, Andy Nguyen, and Carola Wenk. Approximate map matching with respect to the Fréchet distance. In *2011 Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 75–83. SIAM, 2011.

[20] Dua Dheeru and Efi Karra Taniskidou. UCI machine learning repository, 2017.

[21] Anne Driemel, Sariel Har-Peled, and Carola Wenk. Approximating the Fréchet distance for realistic curves in near linear time. *Discrete & Computational Geometry*, 48(1):94–127, Jul 2012.

[22] Fabian Dütsch and Jan Vahrenhold. A filter-and-refinement-algorithm for range queries based on the Fréchet distance (GIS Cup). In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'17, pages 100:1–100:4, New York, NY, USA, 2017. ACM.

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

[24] M Maurice Fréchet. Sur quelques points du calcul fonctionnel. *Rendiconti del Circolo Matematico di Palermo (1884-1940)*, 22(1):1–72, 1906.

[25] GeoLife GPS Trajectories. https://www.microsoft.com/en-us/download/details.aspx?id=52367. Accessed: 2018-12-03.

[26] Joachim Gudmundsson, Majid Mirzanezhad, Ali Mohades, and Carola Wenk. Fast fréchet distance between curves with long edges. *Int. J. Comput. Geom. Appl.*, 29(2):161–187, 2019.

[27] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[28] Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

[29] ACM SIGSPATIAL GIS Cup 2017 Data Set. https://www.martinwerner.de/datasets/san-francisco-shortest-path.html. Accessed: 2018-12-03.

[30] Hong Wei, Riccardo Fellegara, Yin Wang, Leila De Floriani, and Hanan Samet. Multi-level filtering to retrieve similar trajectories under the Fréchet distance. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '18, pages 600–603, New York, NY, USA, 2018. ACM.

[31] Martin Werner and Dev Oliver. ACM SIGSPATIAL GIS Cup 2017 - Range Queries Under Fréchet Distance. *ACM SIGSPATIAL Newsletter, To Appear.*, 2018.

[32] Tim Wylie and Binhai Zhu. Intermittent map matching with the discrete Fréchet distance. *arXiv preprint arXiv:1409.2456*, 2014.

[33] Jianbin Zheng, Xiaolei Gao, Enqi Zhan, and Zhangcan Huang. Algorithm of on-line handwriting signature verification based on discrete Fréchet distance. In Lishan Kang, Zhihua Cai, Xuesong Yan, and Yong Liu, editors, *Advances in Computation and Intelligence*, pages 461–469, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[34] Yu Zheng, Xing Xie, and Wei-Ying Ma. Understanding mobility based on gps data. September 2008.

[35] Yu Zheng, Xing Xie, and Wei-Ying Ma. Mining interesting locations and travel sequences from gps trajectories. April 2009. WWW 2009.

[36] Yu Zheng, Xing Xie, and Wei-Ying Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data(base) Engineering Bulletin*, June 2010.