Walking the Dog Fast in Practice: Algorithm Engineering of the Fréchet Distance

Karl Bringmann
Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
kbringma@mpi-inf.mpg.de

Marvin Künnemann
Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
marvin@mpi-inf.mpg.de

André Nusser
Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
Saarbrücken Graduate School of Computer Science, Saarland Informatics Campus, Germany
anusser@mpi-inf.mpg.de

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-neighbor 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-neighbor data structure.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Design and analysis of algorithms

Keywords and phrases curve simplification, Fréchet distance, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SoCG.2019.17

Related Version A full version of this paper is available at `https://arxiv.org/abs/1901.01504`.

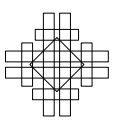
Supplement Material `https://github.com/chaot4/frechet_distance`

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 [28], map matching [15, 27, 16, 10], and clustering [12, 13]. 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

35th International Symposium on Computational Geometry (SoCG 2019).
Editors: Gill Barequet and Yusu Wang; Article No. 17; pp. 17:1–17:21

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



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 [21], 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 $\epsilon > 0$ [6]. Even for realistic models of input curves, such as c-packed curves [18], exact computation of the Fréchet distance requires time $n^{2-o(1)}$ under SETH [6]. 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 [18, 7].

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 [25]. 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-neighbor problem*.

The three top implementations [5, 11, 19] 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 [5] 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 [6, 8]. 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 [22] 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 in the full version 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-neighbor 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 [24], a set of handwritten characters called the Character Trajectories Data Set [26] from [17], and the GeoLife data set [3] of Microsoft Research [30, 29, 31]. 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 [5], Buchin et al. [11], and Dütsch and Vahrenhold [19]. We compare our implementation with all of them by running their implementations on our new benchmark set for the near-neighbor problem and also comparing to the improved decider of [5]. The comparison shows significant speed-ups up to almost a factor of 30 for the near-neighbor problem and up to more than two orders of magnitude for the decider.

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 [9]. This relies on the Four Russians technique and is mostly of theoretical interest. There are many variants of the Fréchet distance, e.g., the discrete Fréchet distance [1, 20]. After the GIS Cup 2017, several practical papers studying aspects of the Fréchet distance appeared [4, 14, 23]. Some of this work [4, 14] 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 [23] 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.

[1] Code and benchmarks are available at: `https://github.com/chaot4/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."

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

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

We call the π_p with $p \in [1, n]$ the *points* on π. A subcurve of π which starts at point p on π 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

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

where $\mathcal{T}_k$ is the set of monotone and continuous 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 $d_F(\pi, \sigma) \leq \delta$ 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 $d_F(\pi, \sigma) \leq \delta$. (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 $d_F(\pi, \sigma) \leq \delta$.

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 $\|\pi_{1 \ldots i}\|$ for every $i \in \{2, \ldots, n\}$. We can then compute the curve length for two indices $i < i'$ on π by $\|\pi_{i \ldots i'}\| = \|\pi_{1 \ldots i'}\| - \|\pi_{1 \ldots i}\|$.

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.

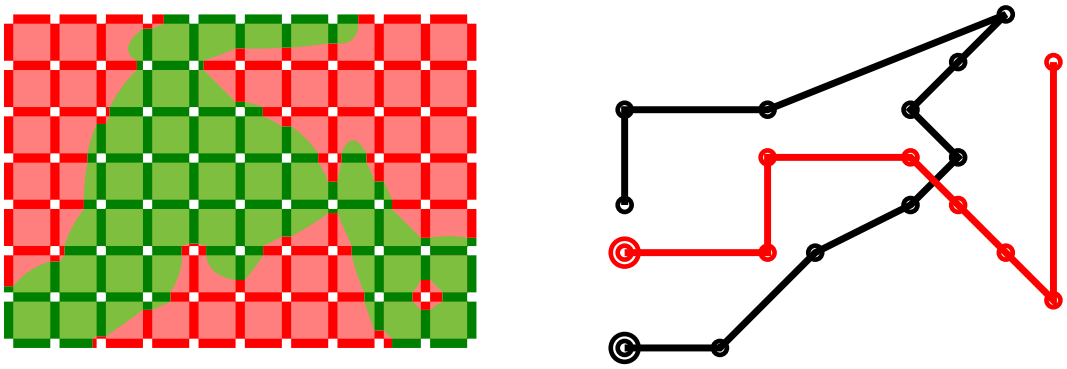


Figure 1 Example of a free-space diagram for curves π (black) and σ (red). The doubly-circled vertices mark the start. The free-space, i.e., the pairs of indices of points which are close, is colored green. The non-free areas are colored red. The threshold distance δ is roughly the distance between the first vertex of σ and the third vertex of π.

3 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 [5] 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.

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 $(i,j) \in \{1,\dots,n-1\} \times \{1,\dots,m-1\}$ is defined as $C_{i,j} := R \cap [i,i+1] \times [j,j+1]$. This is a natural choice as given $C_{i-1,j}$ and $C_{i,j-1}$, we can compute $C_{i,j}$ in constant time; this follows from the simple fact that $F \cap [i,i+1] \times [j,j+1]$ 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].

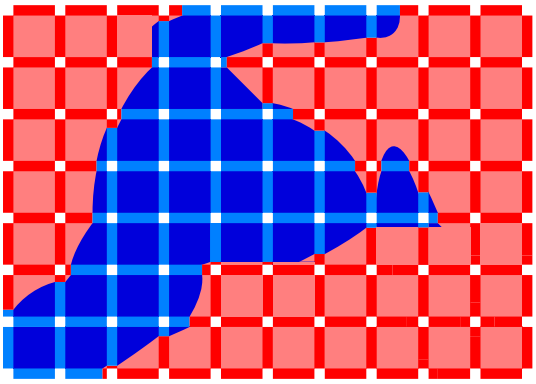


Figure 2 Reachable space of the free-space diagram in Figure 1. The reachable part is blue and the non-reachable part is red. Note that the reachable part is a subset of the free-space.

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 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.

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 [6]. 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 $\emptyset$.

Algorithm 1 Recursive Decider of the Fréchet Distance.

```
1:  procedure DecideFréchetDistance(π, σ)
2:      ComputeOutputs(π, σ, [1, n] × [1, m])
3:      return [(n, m) ∈ R]

4:  procedure ComputeOutputs(π, σ, B = [i, i'] × [j, j'])
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                    ▷ split horizontally
11:                 B_1 = [i, i'] × [j, ⌊(j + j')/2⌋]
12:                 B_2 = [i, i'] × [⌊(j + j')/2⌋, j']
13:             else                                       ▷ split vertically
14:                 B_1 = [i, ⌊(i + i')/2⌋] × [j, j']
15:                 B_2 = [⌊(i + i')/2⌋, i'] × [j, j']
16:         ComputeOutputs(π, σ, B_1) and ComputeOutputs(π, σ, B_2)
```

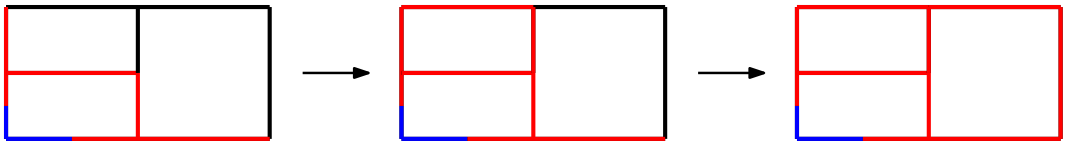


Figure 3 Output computation of a box when inputs are empty. First we can compute the outputs of the top left box and then the outputs of the right box. In this example, we then know that the curves have a Fréchet distance greater than δ as (n, m) is not reachable.

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.

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 section. 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

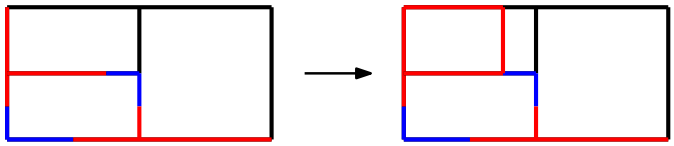


Figure 4 This is an example of shrinking a box in case one of the inputs is empty and the other one starts with an empty part. In this example the top left box has an empty input on the left and the start of the bottom input is empty as well. Thus, we can shrink the box to the right part.

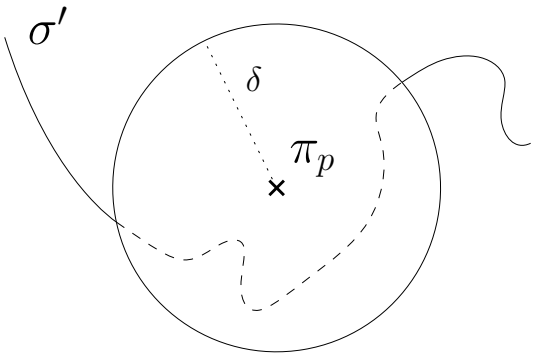


Figure 5 Example of a point π_p and a curve σ' which lead to a simple boundary.

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 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 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 $\{i_{\min}\} \times [j, j']$.

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

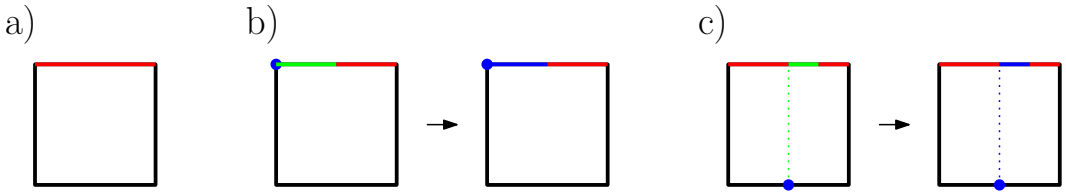


Figure 6 Visualization of the rules for computing outputs using simple boundaries. All three cases are visualized with the top boundary being simple. In a) the boundary is non-free and therefore no point on it can be reachable. In b) the boundary's beginning is free and reachable, enabling us to propagate 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.

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 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 [5]. 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 $\pi' := \pi_{i\dots i'}$ and $\sigma' := \sigma_{j\dots j'}$, 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

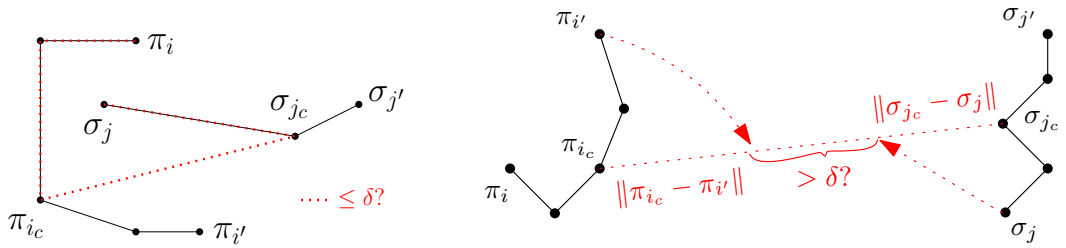
$$\max\{\|\pi_{i\dots i_c}\|, \|\pi_{i_c\dots i'}\|\} + \|\pi_{i_c} - \sigma_{j_c}\| + \max\{\|\sigma_{j\dots j_c}\|, \|\sigma_{j_c\dots j'}\|\} \leq \delta.$$

The triangle equality 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

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

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



(a) HeurClose **(b)** HeurFar

Figure 7 Visualizations of heuristic checks HeurClose and HeurFar.

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 $\sigma_{q\ldots q'}$ 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 $\sigma_{q\ldots q'}$ 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.

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 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

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

```
1:  procedure isSimpleBoundary(π_p, σ_{q...q'})
2:      if HeurFar(π_p, σ_{q...q'}, δ) or HeurClose(π_p, σ_{q...q'}, δ) then
3:          return 'simple'
4:      C ← { {σ_q}   , if ‖p − σ_q‖ ≤ δ                ▷ set of change points
            { ∅       , otherwise
5:      s ← 1, j ← q
6:      while j < q' do
7:          if HeurClose(π_p, σ_{j...j+s}, δ) then
8:              j ← j + s
9:              s ← 2s
10:         else if HeurFar(π_p, σ_{j...j+s}, δ) then
11:             j ← j + s
12:             s ← 2s
13:         else if s > 1 then
14:             s ← s/2
15:         else
16:             P ← {j' ∈ (j, j + 1] | ‖π_p − σ_{j'}‖ = δ}
17:             C ← C ∪ P
18:             j ← j + 1
19:             if |C| > 2 then
20:                 return 'not simple'
21:     return 'simple'
```

with only twelve boxes, i.e., twelve recursive calls, for curves with more than 50 vertices 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

The decider can be divided into two parts:

1. Filters
2. Complete decider via free-space exploration

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 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 adaptive negative filter. Due to space constraints, the detailed explanation of the filters is in Section 4 of the full version.

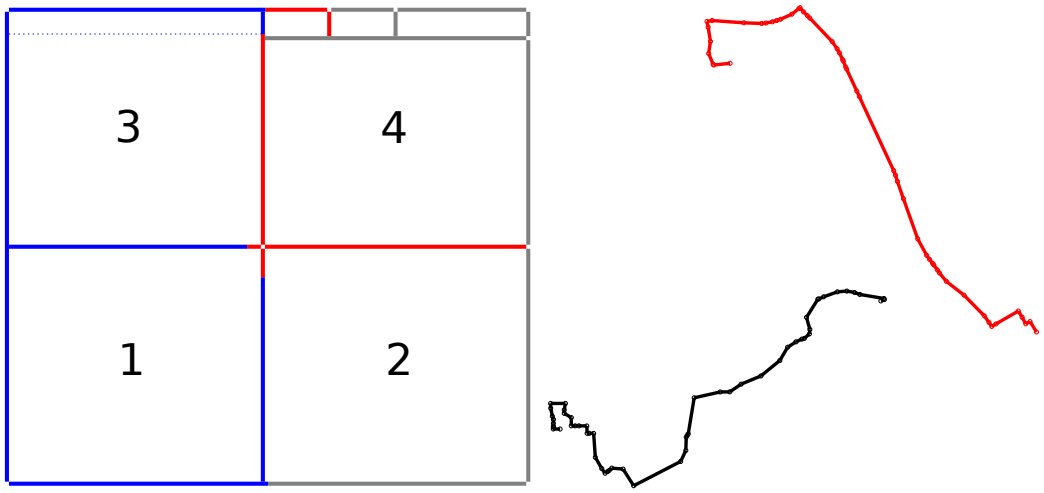


Figure 8 A free-space diagram as produced by our final implementation (left) with the corresponding curves (right). The curves are taken from the SIGSPATIAL dataset. We number the boxes in the third level of the recursion from 1 to 4.

5 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 [5]. 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) [24], the handwritten characters (CHARACTERS) [26], and the GeoLife data set (GEOLIFE) [3]. For all experiments, we used a laptop with an Intel i5-6440HQ processor with 4 cores and 16GB of RAM. See Section 7 of the full version for additional experiments.

5.1 Decider setting

In this section we test the running time performance of our new decider algorithm. 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 $\sigma_1, \ldots, \sigma_n$. For all $k \in \{1, \ldots, \lfloor \log n \rfloor\}$, we

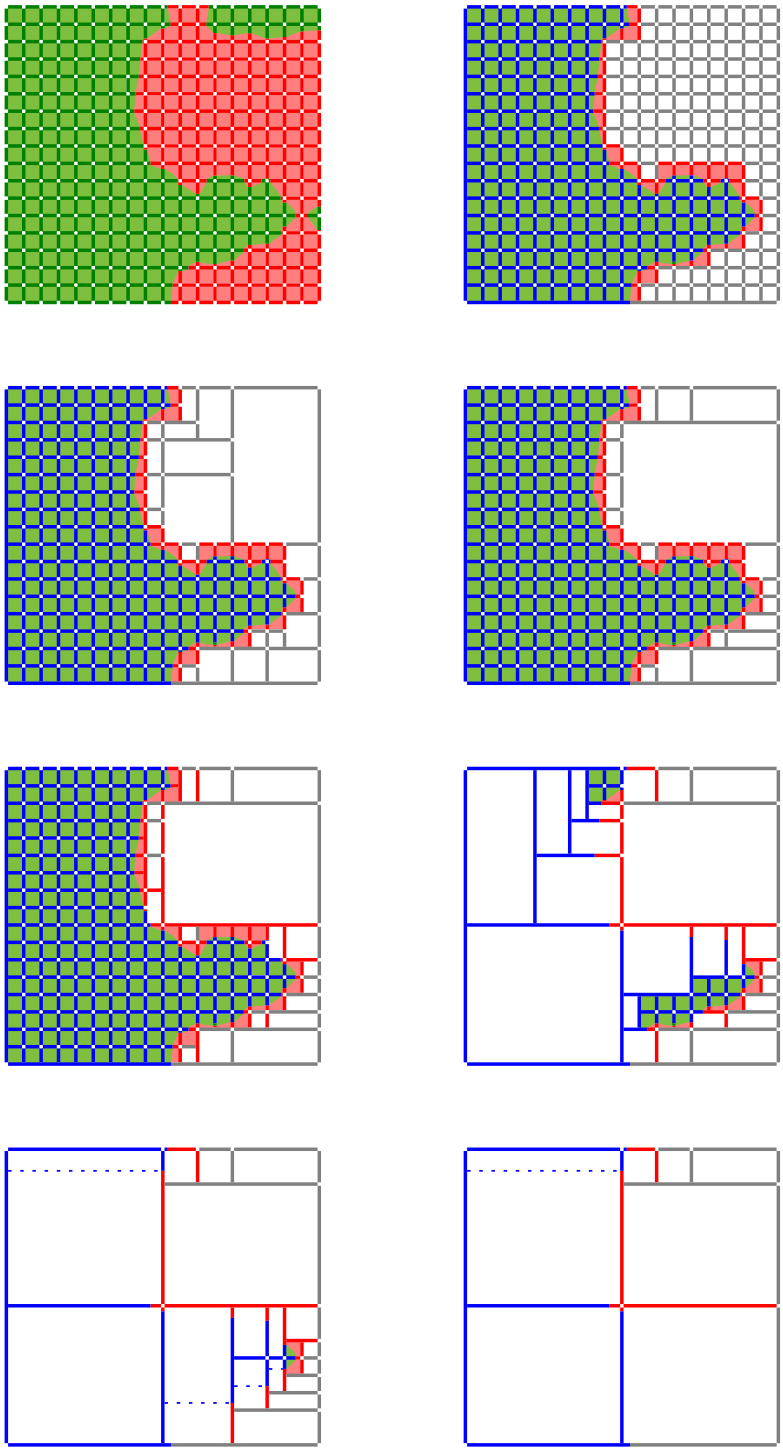


Figure 9 A decider example introducing the pruning rules one by one. They are introduced from top to bottom and left to right. The images in this order depict: the free-space diagram, the reachable space, after introducing Rule I, Rule II, Rule IIIa, Rule IIIb, Rule IIIc, Rule IV, and finally the free-space diagram with all pruning rules enabled. The curves of this example are shown in Figure 8.

- 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 10 we depict timings for running our implementation on the benchmark for all data sets. We can see that distances larger than the exact Fréchet distance are harder than smaller 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 (π, σ_i, δ) when i is smaller. 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 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 [5], on our benchmark and compare it with the running time of our implementation. See Figure 11 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 [5]; 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 12.

5.2 Query setting

Our query data structure is influenced by [5]. We first use spatial hashing (via a kd-tree) on endpoints and extrema of the curves to select a good set of candidate curves, and then use our decider from Section 4 to decide those candidates. See Section 5 of the full version.

Benchmark. We build a query benchmark similar to the one used in [5]. 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.

[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 $\{\sigma_{2^k}, \ldots, \sigma_n\}$.

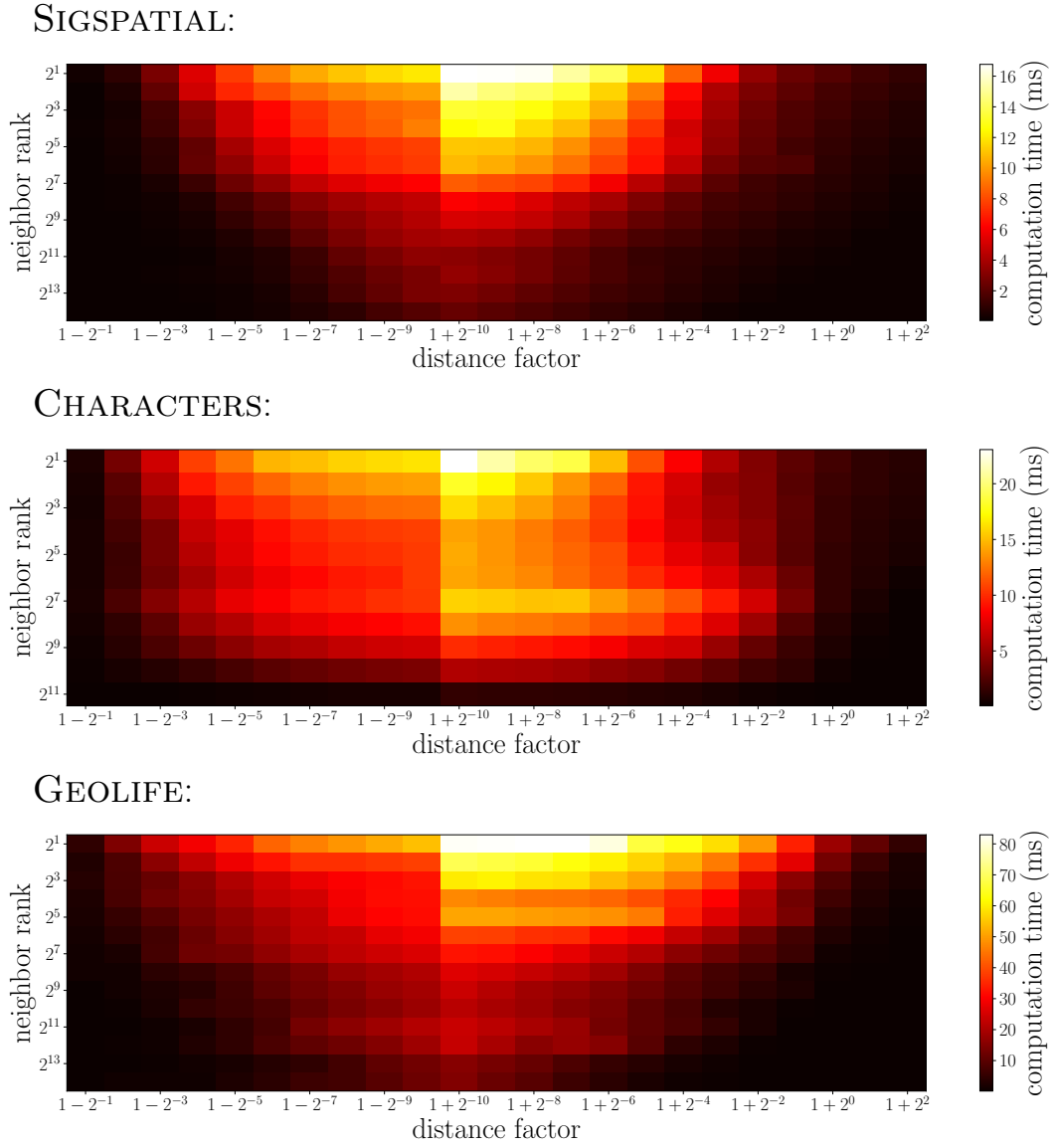


Figure 10 Running times of the decider benchmark when we run our implementation on it.

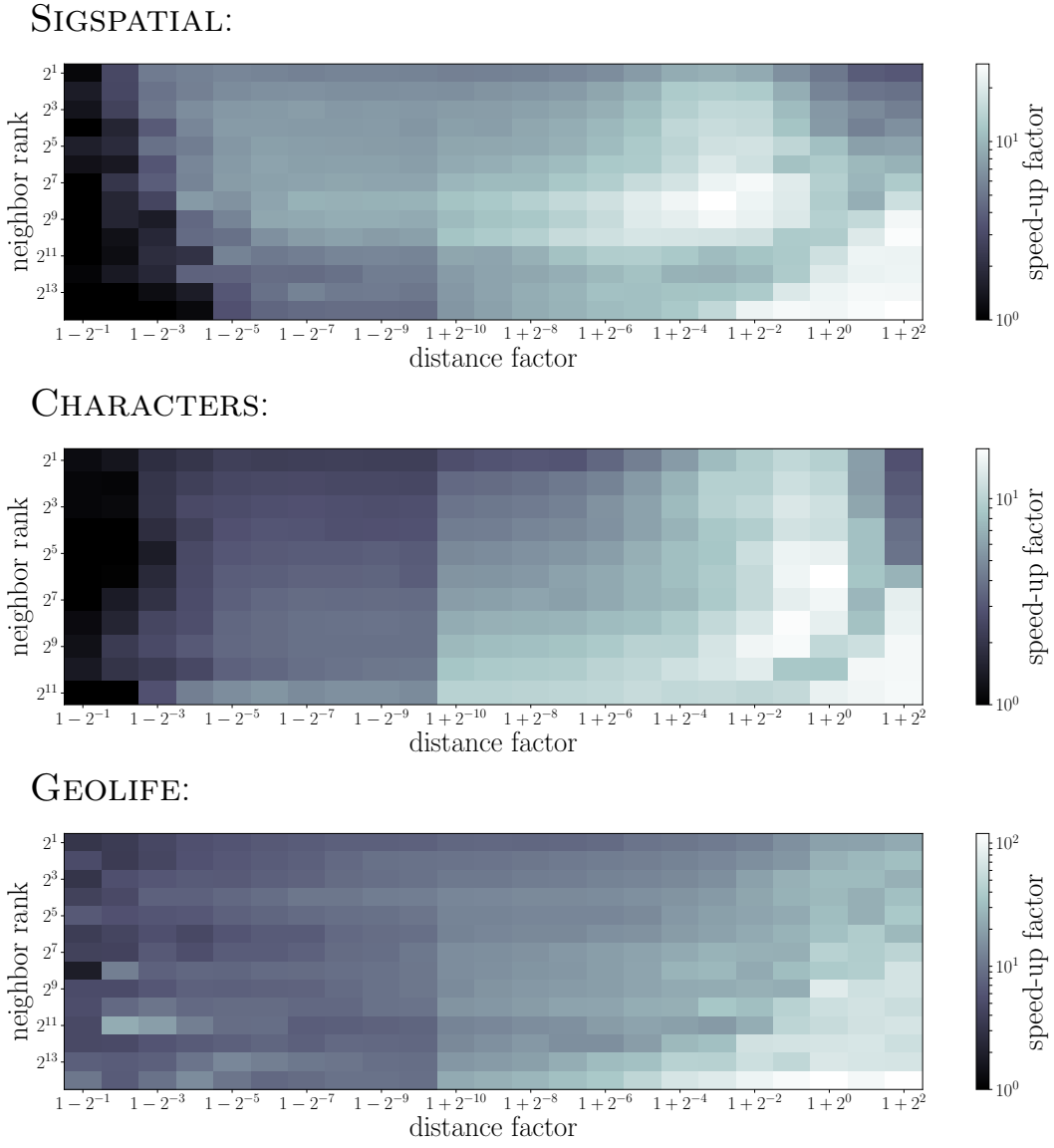


Figure 11 The speed-up factors obtained over the GIS Cup winner on the decider benchmark.

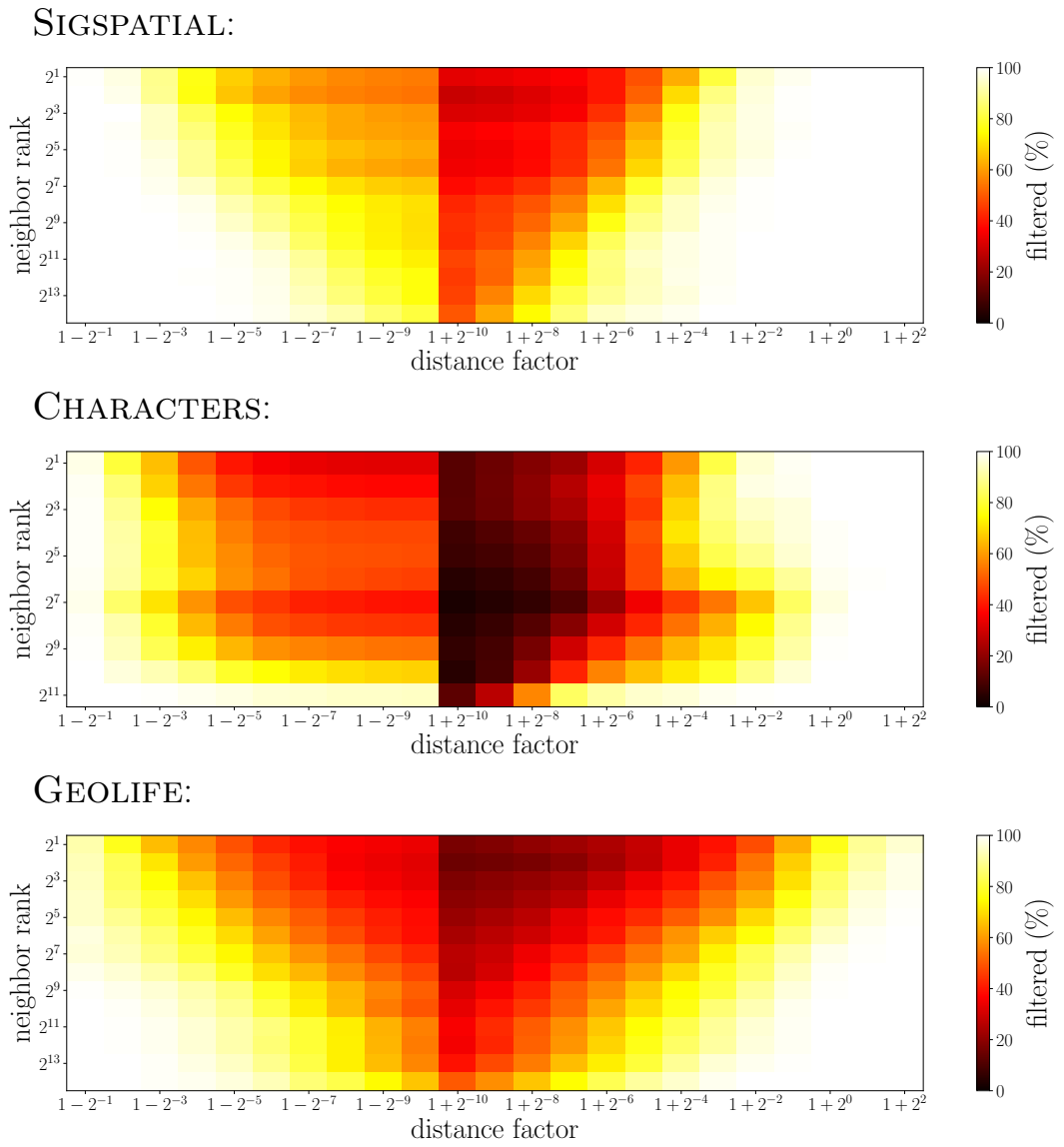


Figure 12 The percentage of queries that are decided by the filters on the decider benchmark.

Table 1 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).

	Sigspatial					Characters					GeoLife				
k	0	1	10	100	1000	0	1	10	100	1000	0	1	10	100	1000
[5]	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
[11]	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
[19]	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 2 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 1 as those are parallelized timings and additionally the timing itself introduces some overhead.

	Sigspatial					Characters					GeoLife				
k	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
adaptive 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

Running times. We compare our implementation with the top three implementations of the GIS Cup on this benchmark. The results are shown in Table 1. Again the running 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 2. 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.

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. doi:10.1137/130920526.

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

4 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.

5 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. doi:10.1145/3139958.3140062.

6 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.

7 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. doi:10.1142/S0218195917600056.

8 Karl Bringmann and Wolfgang Mulzer. Approximability of the discrete Fréchet distance. *Journal of Computational Geometry*, 7(2):46–76, December 2015. doi:10.20382/jocg.v7i2a4.

9 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. doi:10.1007/s00454-017-9878-7.

10 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.

11 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. doi:10.1145/3139958.3140064.

12 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. arXiv:1805.01547.

13 Jonathan Campbell, Jonathan Tremblay, and Clark Verbrugge. Clustering Player Paths. In *FDG*, 2015.

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

15 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.

16 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.

17 Dua Dheeru and Efi Karra Taniskidou. UCI machine learning repository, 2017. URL: http://archive.ics.uci.edu/ml.

18 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, July 2012. doi:10.1007/s00454-012-9402-z.

19 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. doi:10.1145/3139958.3140063.

20 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.

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

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

23 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. doi:10.1145/3274895.3274978.

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

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

26 Ben H Williams. Character Trajectories Data Set. https://archive.ics.uci.edu/ml/datasets/Character+Trajectories. Accessed: 2018-12-03.

27 Tim Wylie and Binhai Zhu. Intermittent Map Matching with the Discrete Fréchet Distance. *arXiv preprint*, 2014. arXiv:1409.2456.

28 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.

29 Yu Zheng, Xing Xie, and Wei-Ying Ma. Understanding Mobility Based on GPS Data. In *Proceedings of the 10th ACM conference on Ubiquitous Computing (Ubicomp 2008)*, September 2008. URL: https://www.microsoft.com/en-us/research/publication/understanding-mobility-based-on-gps-data/.

30 Yu Zheng, Xing Xie, and Wei-Ying Ma. Mining Interesting Locations and Travel Sequences From GPS Trajectories. In *Proceedings of International conference on World Wide Web 2009*, April 2009. WWW 2009. URL: `https://www.microsoft.com/en-us/research/publication/mining-interesting-locations-and-travel-sequences-from-gps-trajectories/`.

31 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.