The Fréchet Distance Unleashed: Approximating a Dog with a Frog

December 3, 2024

Abstract

We show that a variant of the continuous Fréchet distance between polygonal curves can be computed using essentially the same algorithm used to solve the discrete version. The new variant is not necessarily monotone, but this shortcoming can be easily handled via refinement.

Combined with a Dijkstra/Prim type algorithm, this leads to a realization of the Fréchet distance (i.e., a morphing) that is locally optimal (aka locally correct), that is both easy to compute, and in practice, takes near linear time on many inputs. The new morphing has the property that the leash is always as short-as-possible. These matchings/morphings are more natural, and are better than the ones computed by standard algorithms – in particular, they handle noise more graciously. This should make the Fréchet distance more useful for real world applications.

We implemented the new algorithm, and various strategies to obtain fast practical performance. We performed extensive experiments on our new algorithm, and released publicly available (and easily installable and usable) `Julia` and `Python` packages. In particular, the `Julia` implementation, for computing the regular Fréchet distance, seems to be ***significantly faster*** than other currently available implementations. See Figure 2.1.

Links and information about our implementations are available here: frechet.xyz, with numerous detailed examples with animations are available here.

1. Introduction

1.1. Definitions

Given two polygonal curves, their Fréchet distance is the length of a leash that a person needs, if they walk along one of the curves, while a dog connected by the leash walks along the other curve, assuming they synchronize their walks so as to minimize the length of this leash. (I.e. they walk so as to minimize their maximum distance apart during the walk.) Our approach is slightly different than the standard approach, and we define it carefully first.

1

1.1.1. Free space diagram and morphings

Definition 1.1. For a (directed) curve $\pi \subseteq \mathbb{R}^d$, its ***uniform parameterization*** is the bijection $\pi : [0, \|\pi\|] \to \pi$, where $\|\pi\|$ is the length of π, and for any $x \in [0, \|\pi\|]$, the point $\pi(x)$ is at distance x (along π) from the starting point of π.

Definition 1.2. The *free space diagram* of two curves π and σ is the rectangle $R = R(\pi, \sigma) = [0, \|\pi\|] \times [0, \|\sigma\|]$. Specifically, for any point $(x, y) \in R$, we associate the ***elevation function*** $e(x, y) = \|\pi(x) - \sigma(y)\|$.

The free space diagram R is partitioned into a non-uniform grid, where each cell corresponds to all leash lengths when a point lies on a fixed segment of one curve, and the other lies on a fixed segment of the other curve, see Figure 1.1. The sublevel set of the elevation function inside such a grid cell is a clipped ellipse.

Definition 1.3. A ***morphing*** m between π and σ is a (not self-intersecting) curve $m \subseteq R(\pi, \sigma)$ with endpoints $(0, 0)$ and $(\|\pi\|, \|\sigma\|)$. The set of all morphings between π and σ is $\mathcal{M}_{\pi,\sigma}$. A morphing that is a segment inside each cell of the free space diagram that it visits, is *well behaved*[1].

Intuitively, a morphing is a reparameterization of the two curves, encoding a synchronized motion along the two curves. That is, for a morphing $m \in \mathcal{M}_{\pi,\sigma}$, and $t \in [0, \|m\|]$, this encodes the configuration, with a point $\pi\big(x(m(t))\big) \in \pi$ matched with $\sigma\big(y(m(t))\big) \in \sigma$. The ***elevation*** of this configuration is $e(t) = e(m(t)) = \left\|\pi\big(x(m(t)) - \sigma\big(y(m(t))\big)\right\|$.

1.1.2. Fréchet distance

Definition 1.4. The ***width*** of a morphing m between π and σ is $\omega(m) = \max_{t \in [0, \|m\|]} e(t)$. The ***Fréchet distance*** between the two curves π and σ is

$$\mathsf{d}_{\mathcal{F}}(\pi, \sigma) = \min_{m \in \mathcal{M}^+_{\pi,\sigma}} \omega(m),$$

where $\mathcal{M}^+_{\pi,\sigma} \subseteq \mathcal{M}_{\pi,\sigma}$ is the set of all x/y-monotone morphings.

Conceptually, the Fréchet distance is the problem of computing the minimum bottleneck matching between two curves, respecting the order and continuity of the curves.[2] Alternatively, it is an L_∞-norm type measure of the similarity between two curves. It thus suffers from sensitivity to outliers. Furthermore, even if only a small portion of the morphing requires a long leash, the measure, and the algorithms computing it, may use this long leash in large portions of the walk, generating a matching that is loose in many places, see Figure 1.2.

Observe that the Fréchet distance is the minimum value such that the sublevel set of the elevation function has an x/y-monotone path from $(0, 0)$ to $(\|\pi\|, \|\sigma\|)$ in R.

---

[1] All the morphings we deal with in this paper are well behaved.

[2] Formally, since the reparameterization is not one-to-one, this is not quite a matching. One can restrict to using only such bijections, with no adverse effects, but it adds a level of tediousness, which we avoid for the sake of simplicity of exposition.
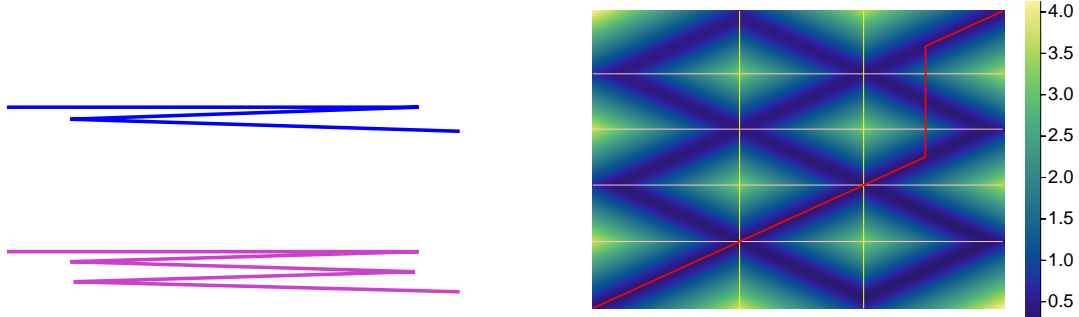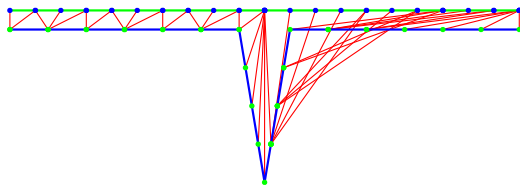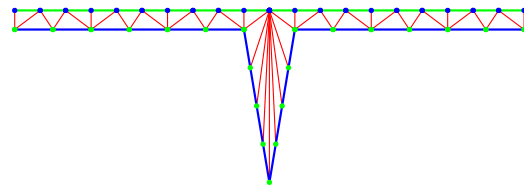
Figure 1.1: Two curves, their free space diagram (and the associated elevation function), and the optimal Fréchet morphing between the two curves encoded as an x/y-monotone curve. More illustrations and animations of this example are available here.

The classical (discrete) Fréchet morphing, caring only about the maximum leash length.

The retractable discrete Fréchet morphing, using the shortest leash possible at each point.

Figure 1.2: A comparison between the classical and retractable Fréchet distances. Observe that the morphing generated by the classical distance can be quite loose in many places. An animation of both morphings is available here.

1.2. Background

Alt and Godau [AG95] presented a rather involved $O(n^2 \log n)$ time algorithm to compute the Fréchet distance using parametric search. The parametric search can be removed by using randomization, giving a simpler algorithm as shown in [HR14]. Buchin *et al.* [BBL+16] presented an alternative algorithm for computing the Fréchet distance that replaces the decision procedure by using a data-structure to maintain appropriate lower envelopes. Despite some simplifications, all these algorithms are somewhat involved.

Unfortunately, it is believed this problem requires quadratic time in the worst case, although a logarithmic speedup is possible; see [BBMM17] and references therein. The quadratic time can be improved for realistic inputs by assuming that the input is "nice", and introducing approximation, but the resulting algorithms are still not simple [DHW12].

1.2.1. Variants of the Fréchet distance

Weak Fréchet distance. The ***weak Fréchet distance*** allows morphings where the agents are allowed to go back and traverse portions of the curve visited previously (i.e., the morphing does not have to be x/y-monotone). Since the weak Fréchet distance allows considering more parameterizations, it is potentially smaller, and we have that $\mathsf{d}_{\mathcal{F}}^w(\pi, \sigma) \leq \mathsf{d}_{\mathcal{F}}(\pi, \sigma)$, for any two curves π, σ. Elegantly, Alt and Godau [AG95] showed the weak Fréchet distance can be reduced to computing the minimum spanning tree of an appropriate graph. Unfortunately, there does not seem to be a natural way to overcome this non-monotonicity (and thus get the "strong" version).

Discrete Fréchet distance. The complexity of these algorithms, together with sensitivity of the Fréchet distance to noise, lead to using "easier" related measures, such as the discrete version of the problem, and dynamic time-warping (discussed below). In the discrete version, you are given two sequences of points $p_1, \ldots, p_n$, and $q_1, \ldots, q_m$, and the purpose is for two "frogs" starting at p_1 and q_1, respectively, to jump through the points in the sequence until reaching p_n, q_m, respectively, while minimizing the maximum distance between the two frogs during this traversal. At each step, only one of the frogs can jump from its current location to the next point in its sequence (no jumping back). Computing the optimal distance under this measure can be done by dynamic-programming, similar to the standard approach to edit-distance. Indeed, the configuration space here is the grid $H = [\![n]\!] \times [\![m]\!]$, where $[\![n]\!] = \{1, \ldots, n\}$.

To use the discrete version in the continuous case, one sprinkles enough points along both input curves, and then solves the discrete version of the problem. Beyond the error this introduces, to get a distance that is close to the standard Fréchet distance, one has to sample the two curves quite densely in some cases.

For the (monotone) discrete Fréchet distance the induced graph on the grid H is a DAG, and the task of computing the Fréchet distance is to find a minimum bottleneck path from $(1, 1)$ to (n, m), where the weights are on the vertices. Here, the weight on the vertex (i, j) is the distance $\|p_i - q_j\|$. In particular, a Fréchet morphing is an x/y-monotone path in H

from $(1, 1)$ to (n, m). The standard algorithm to do this traverses the grid, say, by increasing rows i, and in each row by increasing column j, such that the value at (i, j) is the maximum of the length of the leash of this configuration, together with the minimum solution for $(i − 1, j)$ and $(i, j − 1)$. This algorithm leads to a straightforward, $O(nm)$ time algorithm for the discrete Fréchet distance. However, the morphing computed might be inferior, see Figure 1.2 for such a bad example.

Retractable Fréchet.   For simplicity, assume that the pairwise distances between all pairs of points in the two sequences are unique. We would like to imagine that we have a retractable leash, that can become shorter at times, and the leash "resists" being longer than necessary. It is thus natural to ask for a morphing where the leash as short as possible at any point in time.

Informally, the optimal *retractable Fréchet morphing* between the two sequences includes the bottleneck configuration, realizing the Fréchet distance, in the middle of its path, and the two subpaths from the endpoints to this configuration have to be also recursively optimal. This is formally defined and described in Section 3.2. This concept was introduced by Buchin *et al.* [BBMS19]. Interestingly, they show that the discrete version can be computed in $O(nm)$ time, but unfortunately, the algorithm is quite complicated. They also show that the continuous retractable Fréchet can be computed in $O(n^3 \log n)$ time.

Buchin *et al.* [BBMS19] refers to this Fréchet distance as *locally correct*, but we prefer the *retractable* labeling. The term "retractable Fréchet" was used by Buchin *et al.* [BBL+16], but in a different (and not formally defined) context than ours.

(Continuous) Dynamic Time Warping.   One way to get less sensitivity for noise is to compute the total area "swept" by the leash as the walk is being performed. In the discrete case, we simply add up the lengths of the leashes during the configurations in the walk. There is also work on extending this to the continuous setting [MP99, BBK+20]. For the continuous case this intuitively boils down to computing (or approximating) an integral along the morphing. See Section 5.5 for more details.

1.2.2. Critical events

The standard algorithm for computing the Fréchet distance works by performing a "binary" search for the Fréchet distance. Given a candidate distance, it constructs a "parametric" diagram that is a grid, where inside each grid cell the feasible region is a clipped ellipse. The task is then to decide if there is an x/y-monotone path from the bottom left corner to the top right corner, which is easily doable. The critical values the search is done over are:

(I) *Vertex-vertex events*: The distance between two vertices of the two curves,

(II) *Vertex-edge events*: The distance between a vertex of one curve, and an edge of the other.

(III) *Monotonicity events*: This is the minimum distance between a point on one edge e of the curves, and (maximum distance to) two vertices u, v of the other curve. Specifically,

it is realized by the point on e with equal distance to u and v.

The first two types of events are easy to handle, but the monotonicity events are the bane of the algorithms for the Fréchet distance.

1.2.3. Algorithm engineering the Fréchet distance

Given the asymptotic complexity and involved implementations of the aforementioned algorithms, there has been substantial work on practical aspects of computing the Fréchet distance. In particular, in 2017, ACM SIGSPATIAL GIS Cup had a programming challenge to implement algorithms for computing the Fréchet distance. See [WO18] for details.

 More recently, Bringmann *et al.* [BKN21] presented an optimized implementation of the decider for the Fréchet distance. Somewhat informally, Bringmann *et al.* [BKN21] builds a decider for the Fréchet distance using a *kd*-tree over the free space diagram, keeping track of the reachable regions on the boundary of each cell, refining cells by continuing down the (virtual) *kd*-tree if needed.

1.3. Our results

1.3.1. Result I: A new algorithm for retractable discrete Fréchet

We observe that a natural approach to compute the retractable Fréchet morphing is to modify Dijkstra's/Prim's algorithm so that it solves the minimum bottleneck path problem. This observation surprisingly seems to be new, and leads to a simpler (but log factor slower) algorithm for computing it. The only modification of Dijkstra necessary is that one always handles the cheapest edge coming out of the current cut induced by the set of vertices already handled. (In the discrete Fréchet case, the weights are on the vertices, but this is a minor issue.) This modified version of Dijkstra is well known, but we include, for the sake of completeness, the proof showing that it indeed computes the recursively optimal path, which is also a retractable Fréchet morphing between the two sequences. This immediately leads to better and more natural morphings, see Figure 1.2.

 Maybe more importantly, in practice, one does not need to explore the whole space of nm configurations (since we are in the discrete case, a configuration $(i, j) \in [\![n]\!] \times [\![m]\!]$ encodes the matching of p_i with q_j), as the algorithm can stop as soon as it arrives at the destination configuration (n, m). Informally, if the discrete Fréchet distance is "small" compared to the vast majority of pairwise distances (i.e., the two sequences are similar), then the algorithm only explores a small portion of the configuration space. Thus, this leads to an algorithm that is faster than the standard algorithm in many natural cases, while computing a significantly better output morphing.

1.3.2. Result II: A new distance and algorithm: **VE**-Fréchet

It is not clear how to extend the above to the continuous case. A natural first step is to consider the continuous Fréchet distance, where one restricts the solution inside each cell
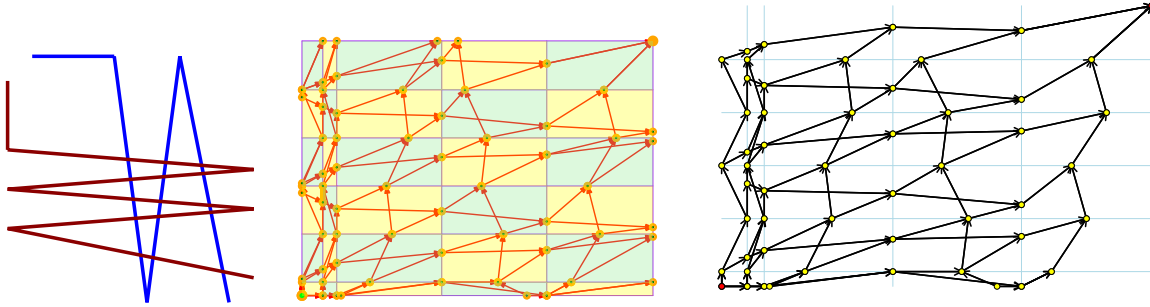
Figure 1.3: Two curves, and their associated VE-Fréchet graph. Note, that every internal edge of the grid contains a portal (i.e., vertex of the graph), but in many cases the portals of two edges are co-located on their common vertex. For the edges adjacent to the starting and ending corners we set their portals to lie at the corners themselves. See here for more info.

of the free space diagram to be a segment (which is already the case for the morphings computed by existing algorithms), but more importantly, insisting that the shape of this segment must be determined only locally, thus facilitating a greedy strategy compatible with the retractable approach. In practical terms, we throw away the (global) monotonicity events.

Traveling only through vertex-edge events. Because of the continuity and strict convexity of the elevation function, the function has a unique minimum on each edge of the free space diagram grid – geometrically, this is the minimum distance between a vertex of one curve, and an edge of the other curve (a *vertex-edge event*). We restrict our solution to enter and leave a cell only through these *portals* (which are easy to compute). The continuous configuration space now collapses to a discrete graph that is somewhat similar to the natural grid graph on $[\![n]\!] \times [\![m]\!]$. Indeed, a grid cell has four portals on its boundary edges. Specifically, there are directed edges from the portal on the bottom edge, to the portal on the top and right edges of the cell. Similarly, there are edges from the portal on the left edge, to the portals lying on the top and right edges. As before, the values are on the portals, and our purpose is to compute the optimal bottleneck path in this grid-like graph. Examples of this graph are depicted in Figure 1.3, Figure 4.1 and Figure 4.2.

Remark 1.5. A somewhat similar idea was used by Munich and Perona [MP99], but they used it in the other direction – namely, in defining a better CDTW distance for two discrete sequences. However, this idea was already present (implicitly) in the original work of Alt and Godau [AG95] – indeed, their algorithm for the Weak Fréchet distance uses only the Vertex-Edge events (i.e., edges in the free space diagram). This boils down to solving the bottleneck shortest path problem in an undirected graph. In this case, this problem can be solved by computing the minimum spanning tree (e.g., by Prim's algorithm, which is a variant of Dijkstra's algorithm), as Alt and Godau do. For the directed case, one needs

to use a variant of Dijkstra's algorithm [GT88] – see Lemma 3.2. See also Buchin *et al.* [BBD+17] who also used a similar idea.

We can now run the retractable bottleneck shortest-path algorithm (i.e., the variant of Dijkstra described above) on this implicitly defined graph computing the vertices and edges of it, as they are being explored. For many natural inputs, this algorithm does not explore a large fraction of the configuration space, as it involves distances that are significantly larger than the maximum leash length needed. The algorithm seems to have near linear running time for many natural inputs. The **VE-Fréchet** morphing is the one induced by the computed path in this graph. Unfortunately, the VE-Fréchet morphing might allow the agents to move backwards on an edge, but importantly, the motion across a vertex is monotone. Namely, the VE-Fréchet is monotone for vertices, but not necessarily monotone on the edges. A vertex is thus a *checkpoint* that once passed, cannot be crossed back.

1.3.3. Result III: New algorithm for the regular Fréchet distance

The natural question is how to use the (easily computable) VE-Fréchet morphing to compute the optimal (regular) continuous Fréchet distance. We next describe how this can be done in practice.

The hunt for a monotone morphing. We denote the VE-Fréchet distance between two curves π and σ by $\mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma)$. Clearly, we have that $\mathsf{d}_{\mathcal{F}}^{w}(\pi, \sigma) \leq \mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma) \leq \mathsf{d}_{\mathcal{F}}(\pi, \sigma)$.

One can of course turn any morphing into a monotone one by staying put instead of moving back. This is appealing for VE-Fréchet, as the corresponding VE morphing m, say between two curves π and σ, never backtracks over vertices (only edges), so we already expect the error this introduces to be relatively small. Let m^+ denote the monotone morphing resulting from this simple strategy. Observe that

$$\omega(m) = \mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma) \leq \mathsf{d}_{\mathcal{F}}(\pi, \sigma) \leq \omega(m^+).$$

In particular, if $\omega(m) = \omega(m^+)$, then $\mathsf{d}_{\mathcal{F}}(\pi, \sigma)$ is realized by m^+, and we have computed the Fréchet distance between π and σ.

A less aggressive approach is to introduce new vertices in the middle of the edges of π and σ as to enforce monotonicity. Indeed, clearly, if we refine both curves by repeatedly introducing vertices into them, the VE-Fréchet distance between the two curves converges to the Fréchet distance between the original curves, as introducing a vertex in the middle of an edge does not change the regular Fréchet distance, while preventing the VE-Fréchet morphing from backtracking over this point.

We refer to this process of adding vertices to the two curves as **refinement**. In practice, in many cases, one or two rounds of (carefully implemented) refinement are enough to isolate the maximum leash in the morphing from the non-monotonicity, and followed by the above brute-force monotonization leads to the (practically) optimal Fréchet distance. Even for pathological examples, after a few more rounds of refinement, this process computes the exact
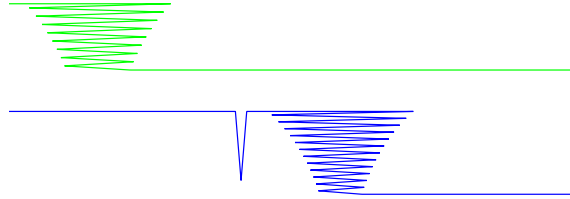
Figure 1.4: For these two curves, the solution involves an agent stopping at one point on one curve while the other agent traverses is zig-zag, and vice versa. The algorithm enforces monotonicity by refining the two curves by introducing new vertices. For the results, see here.

Fréchet distance. That is, the computed lower bound, which is the VE-Fréchet distance, is equal to the width of the computed monotone morphing, which is the Fréchet distance.

Remark 1.6 (Floating point issues). As we are implementing our algorithm using floating point arithmetic, and the calculation of the optimal Fréchet distance involves distances, impreciseness is unavoidable. A slight improvement in preciseness can be achieved by using squared distances (and also slightly faster code) — but for simplicity we have not used this idea in our code. In particular, we take the somewhat pragmatic view, that an approximation to the optimal up to a factor of (say) 1.00001 can be considered as computing the "optimal" solution.

Note that Fréchet morphings are somewhat less sensitive to numerical issues than other geometric problems — indeed, once a morphing is computed, one can compute its width directly.

1.3.4. Result IV: Computing the Fréchet distance quickly for real inputs

The above leaves us with a natural strategy for computing the Fréchet distance between two given curves. Compute quickly, using simplification, a morphing between the two input curves, and maintain (using VE-Fréchet, for example) both upper and lower bounds on the true Fréchet distance. By carefully inspecting the morphing, (re)simplifying the curves in a way that is sensitive to their (local) Fréchet distance, and recomputing the above bounds, one can get an improved morphing. Repeat this process potentially several times till the upper and lower bounds meet, at which point the optimal Fréchet distance has been computed.

This seems somewhat overkill, but it enables us to compute (in practice) the exact Fréchet distance between huge polygonal curves quickly.

1.3.5. Main contribution: Implementation in `Julia` and `Python`

We implemented the above algorithms as official packages in `Python` and `Julia`. A webpage with animations showing our algorithm in action is available here. The `Python` package is available at `https://github.com/eliotwrobson/FrechetLib`, and the `Julia` package is

available at `https://github.com/sarielhp/FrechetDist.jl`. Animations and examples computed by the new algorithm are available at `https://frechet.xyz/`.

1.3.6. Additional results

Sweep distance. We demonstrate how one can convert our algorithm for computing VE-Fréchet to an algorithm that computes a variant of the CDTW distance, which we call the *sweep distance*. One can then use refinement to approximate the CDTW distance. One can also compute a lower bound on this quantity, and the two quantities converge. See Section 5.5 for details.

Fast simplification. We show how to preprocess a curve π with n vertices, in $O(n \log n)$ time, such that given a query w, one can quickly extract a simplification of $\overline{\pi}$ of π, such that $\mathsf{d}_{\mathcal{F}}(\pi, \overline{\pi}) \leq w$. Importantly, the time to extract $\overline{\pi}$ is proportional to its size (i.e., the extraction is output sensitive). More importantly, in practice, this works quite well – the size of $|V(\overline{\pi})|$ is reasonably close (by a constant factor) to the optimal approximation. Combined with greedy simplification, this yields a very good simplification result. See Section 5.6 for details.

1.4. Significance

We demonstrate in this paper that a minor variant of the ***continuous*** Fréchet distance can be computed by a strikingly simple Dijkstra type algorithm. Furthermore, for many real world inputs it runs in near linear time (out of the box). Similarly, for many real world inputs the computed morphing is monotone, thus realizing the (standard) Fréchet distance. More importantly, because of the *retractable* nature of the morphing computed, the matching computed is more natural, and can handle noise/outliers more gracefully than the matchings computed by the current algorithms for the Fréchet distance. Indeed, areas that are noise/outliers, get isolated in the morphing to a small interval, that can be easily identified and handled, see Figure 1.2. We believe that this makes the morphings computed by our new algorithms significantly better for real world applications than previous algorithms.

   We then show how to modify this algorithm to compute the (monotone) Fréchet distance (in cases when the morphing was not already monotone), and how to make it handle large inputs quickly via simplifications (handling all the technical difficulties this gives rise to).

   Finally, we implemented our new algorithms in `Julia` and `Python`, and made them publicly available as standard packages, thus making the computation of the Fréchet distance accessible to a wider audience. Using such packages in `Python`/`Julia` is significantly easier than using any not pre-compiled code in `C++`.

Summary. The combination of simplification, new (and not so new) algorithmic ideas (such as retractablity via Dijkstra, and VE-Fréchet, among others), and careful implementation, leads to fast performance in practice beating other implementations. We discuss our

implementation next — the algorithmic ideas described above are covered in detail in the appendices.

2. Implementation and experiments

To demonstrate the practicality of our techniques, we have developed open-source packages that implement our algorithms. Our implementations follow the ideas described in the paper – aggressively using simplification, and upper/lower bound computations to guarantee we computed the optimal Fréchet distance. Furthermore, because our algorithms use the retractable VE-Fréchet as our building block, the Fréchet morphings our implementation computes look better than what the standard Fréchet algorithms would return.

To see numerous animations and more information, the reader is encouraged to visit the (anonymized) site: `https://frechet.xyz`.

Dataset	# pairs	avg # points	Total RT in seconds			
			`C++`	`Julia`	`Julia` MT	`C++`/`Julia`
sigspatial	$322,000$	247.8	145	102	25	**1.42**
Characters	$253,000$	120.9	90	72	18	**1.25**
Geolife	$322,000$	1080.4	646	70	30	**9.22**
All tests combined			881	244	73	**3.61**

Figure 2.1: `Julia`/`C++` comparison, demonstrating that the `Julia` implementation is significantly faster. The `C++` implementation is from Bringmann *et al.* [BKN21]. Running times are in seconds. `Julia` MT stands for the multi-threaded implementation – since multi-threading is easy in Julia, and the task at hand is easily parallizable, we tested this, but of course this should not be taken as a direct (or remotely fair) comparison to the `C++` implementation, and is provided for the reader amusement. In particular, the multi-threading done is pretty naive, and better performance should be possible by better fine-grained partition of the tasks. The tests were performed on a Linux system with 64GB memory with Intel `i7-11700` CPU, with a decent 16 threads CPU, but far from the fastest hardware currently available. The "# pairs" column is the number of pairs of curves that had their Fréchet distance compared during this test. The "avg # points" is the average number of vertices per curve in this input set – underlying the benefit of simplification for the GeoLife input set.

2.1. Hardware and Implementations

The tests were all performed on a Linux system with 64GB of memory and an Intel `i7-11700` CPU with 16 threads. This system is a fast desktop, but far from the best possible hardware currently available. We were also able to perform development for the experiments on a laptop with substantially weaker hardware, giving anecdotal evidence of the scalability of

our algorithms. We use several real-world datasets featured in prior works to demonstrate the effectiveness of our algorithm.

Here, we discuss the individual implementations used in our experiments. The first two (`Julia` and `Python`) use the algorithmic techniques discussed in this work. The third is by Bringmann *et al.* [BKN21] written in `C++`, which uses completely different algorithmic ideas. We refer to each implementation by language for convenience, although our tasks are meant to evaluate effectiveness of the algorithmic approaches.

2.1.1. Julia

The `Julia` code is about 5000 lines for the library, and an additional 5000 for the examples/testing code. It implements the algorithms described here faithfully, with some additional hacks to handle floating point issues. `Julia` has (surprisingly easy to use) support for multi-threading (unlike `C++` or `Python`), and we tested also a multi-threaded version (this is shown in the MT column in Figure 2.1). The anonymized code is available from `https://frechet.xyz/`, including instructions how to replicate our tests.

The `Julia` package is available at `https://github.com/sarielhp/FrechetDist.jl`. Animations and examples computed by the `Julia` code are available at `https://frechet.xyz/`.

`Julia` low-level optimizations.   We were able to significantly improve the performance of the `Julia` code using some standard low-level optimizations: (i) `Julia` performs inlining code automatically, but explicitly forcing it to inline some low-level key functions (such as computing the distance between two points), improved the performance measurably ($>$ 10%). `Julia` also has high overhead for creating `structs` that get destroyed immediately (essentially because it allocates them on the heap, instead of the stack like `C++`). Thus, by rewriting the code so that computing the distance of a point from a segment is done directly on the endpoints of the segment (where the segment is an edge of a polygon) also improved performance. Caching simplifications, so that one can extract/compute new simplification quickly, among other ideas, see Section 5.6, were useful in creating fast code.

One allocation to rule them all.   Since `Julia` performance suffers when a lot of dynamic allocations are done (because of the automatic garbage collection), it turned out that implementing the retractable VE-Fréchet without using hashing (for the vertices of the graph) resulted in much faster code. To this end, one pre-allocates a quadratic size array representing the vertices of the grid (but exploring only the relevant portions of the grid/this array) — this works because the allocation/initialization of this (single!) array is extremely fast. This seems to yields significant speedups somewhere up to ten times faster than the hashing version, depending on the input (for large inputs, the two implementations seem to have similar running times).

2.1.2. Python

For the `Python` implementation, we represented curves using NumPy arrays [HMW+20], where each row in the array is a single point on the input curve. This representation is memory efficient and allows for fast computations of distances between points using library primitives. We also made heavy use of Numba [LPS15] to further improve performance, as many of the algorithms discussed in this work are iterative and thus easy to accelerate using the library.

Numerical Issues. Although not exclusive to this implementation, the `Python` code suffers from some issues related to numerical precision. Specifically, this implementation struggles with underflow of floating point numbers, which seems to occur in situations where consecutive points are too close in the input data (i.e., the distance between points is too small). However, this doesn't represent a meaningful distance between these points, and this can be resolved by adding simplification as a preprocessing step (such as the algorithm in Section 5.6).

Our experiments are meant to emphasize the performance of our algorithmic techniques, and we do not have any tasks specifically designed to evaluate issues with floating point numbers or methods which can be used to combat this problem. In particular, we did not encounter these issues in any significance with the datasets mentioned in this work. This is a potential avenue for future work if given data that encounters these issues frequently.

2.1.3. C++

The `C++` implementation is provided by Bringmann *et al.* [BKN21], which seems to be the state of the art (they claim to be faster than any previously available implementation). The algorithm uses completely different algorithmic techniques to those of this work, providing a point of comparison for the scalability of our implementations.

Informally, the implementation of Bringmann *et al.* [BKN21] builds a decider for the Fréchet distance using a *kd*-tree over the free space diagram, keeping track of the reachable regions on the boundary of each cell, refining cells by continuing down the (virtual) *kd*-tree if needed.

Note, that unlike Bringmann *et al.* [BKN21], our implementation can compute the ***exact*** Fréchet distance — although in some cases, because of numerical issues the "exact" results are just (high quality) approximations, see Remark 1.6.

2.2. Experiments

To compare the implementations, we performed benchmarks using two different tasks using real-world datasets. These were selected to determine the scalability of the algorithms relative to the large size of real-world data. This section is organized by the different tasks performed.

2.2.1. Direct Fréchet distance computations

To compare the `Julia` and `Python` implementations of our algorithms, we directly computed the Fréchet distance at different levels of approximation using our algorithms.

This was done on individual pairs of curves selected from different datasets. Specifically, we use curves taken from the GeoLife dataset [ZFX+11] (as used in [BKN21]), a stork migration dataset [RKT+18b], (retrieved from Movebank [RKT+18a], used in [BBK+20]), and a pigeon flight dataset [PGR+17]. Figure 2.2 lists the individual curves from each dataset used in our benchmarks.

We do not perform any type of transformations on the input data (to reduce aberration due to Earth's curvature, for example), as we are primarily interested in the execution speed of our algorithm.

See Figure 2.4 and Figure 2.3 for the results (observe that the `Julia` code was tested for 1.001 approximation – we have not done the same for the `Python` code, since it was numerically unstable in this case). The `Python` performance on these datasets is competitive with the `Julia` implementation, although in some cases significantly slower. In general, the `Julia` implementation is faster (we did not perform enough experiments to decide how much faster), but it is clear that the `Python` implementation is fast and robust enough to be used in practice. The runtimes also do not include the Numba JIT compilation times.

Specifically, the slow performance observed on some test cases for the `Python` implementation may be caused by using sub-optimal data structures for some simplification tasks. We hope to improve this performance in further development of this implementation.

2.2.2. Fréchet Decider

We compared our implementation performance in `Julia` to the `C++` implementation. We evaluated our implementations with the same benchmarking task as Bringmann *et al.* [BKN21], using the provided code that downloads and generates Fréchet distance tests. We emphasize that this task and datasets were introduced prior to to this work, and were not selected to be favorable to the techniques introduced here.

This task used curves from three datasets; the dataset from the SIGSPATIAL GIS Cup [WO18], a dataset of handwritten characters (Characters) [Wil08], and the GeoLife dataset [ZFX+11]. Each dataset is composed of numerous input curves, and then a list of pairs of curves together with a threshold, and the goal is to decide whether the Fréchet distance is above/below/equal to this threshold. See Figure 2.1. This task is naturally friendly to simplifications – we implemented a fast simplification data-structure. We iteratively simplify the two curves being compared to progressively finer resolutions, comparing them using the (monotonized) `VE`-Fréchet distance. If the comparison is outside the error interval, the result is returned, otherwise, the algorithm goes to the next finer resolution.

The GeoLife [ZFX+11] dataset benefits tremendously from the simplification strategy, and the `Julia` implementation is more than 9 times faster than the `C++` code, see Figure 2.1. The worst dataset for the `Julia` implementation is the characters dataset [Wil08] where it seems to be about 25% ***faster*** than the `C++` code. This dataset both does not benefit from

simplification, and also seems to give rise to many non-monotone morphings. The Sigspatial seems be somewhat resistant to approximation, and our code is only about 40% faster than the `C++` code in this case. We emphasize that each test in Figure 2.1 involves the execution of well over $> 250,000$ runs of the algorithm for different curves, so these results seems quite representative of real world performance.

As for the general `C++` vs `Julia` performance comparison, this seems to be case dependent, with reported cases where either is faster. The common belief is that a careful enough implementation in `C++` would be faster from `Julia` implementation (i.e., some of `Julia` libraries are implemented in `C++`).

In particular, the significant speed (646 seconds vs 70) improvement in the case of the GeoLife input suggests our algorithm is much faster in cases of longer curves where simplification really helps. Note that while our code in `Julia` uses a single thread, it uses an automatic garbage collector that runs in parallel threads.

2.3. Discussion

Overall, it is clear that our `Julia` implementation is faster than other currently available implementations for computing the Fréchet distance. Beyond that `Julia` seems like a great programming language to develop geometric algorithms — providing a combination of a safe, multi-threaded, high-level programming language with the performance of `C++`, without the pain involved in working in `C++`.

One optimization used by the `Julia` code, that should be generally useful, is the following. As a preprocessing step, precompute a hierarchy of simplified curves for each input curve. This improves the query process, but makes the preprocessing more expensive. Thus, storing such precomputed hierarchies might be a good idea if input curves are going to be used repeatedly for performing distance queries. We emphasize that the reported running times include this (light) preprocessing stage (interestingly, the SIGSPTIAL [WO18] competition allowed such preprocessing to not be included in the overall running time).

Input	Description
1	`birds:   1787_1 / 1797_1`
2	`birds:   2307_3 / 2859_3`
3	`birds:   2322_2 / 1793_4`
4	`GeoLife 20080928160000 / 20081219114010`
5	`GeoLife 20090708221430 / 20090712044248`
6	`Pigeons RH887_1 / RH887_11`
7	`Pigeons C369_5 / C873_6`
8	`Pigeons C360_10 / C480_9`

Figure 2.2: The inputs tested and where they are taken from, `birds` referring to the stork migration dataset [RKT+18a], the GeoLife dataset [ZFX+11], and `Pigeons` referring to a dataset from [PGR+17].

Input	P #	Q #	≈4	≈1.1	≈1.01	≈1.001	Exact	VER
1	10,406	11,821	0.106	0.256	4.164	16.639	0.476	27.743
2	16,324	14,725	0.114	0.428	0.985	4.762	3.522	19.734
3	22,316	4,613	0.151	0.651	1.207	3.262	0.882	7.138
4	56,730	91,743	0.578	1.207	4.215	28.109	4.259	---
5	6,103	9,593	0.043	0.040	0.305	0.519	0.484	5.678
6	2,702	1,473	0.034	0.084	1.367	1.387	0.160	1.081
7	1,068	1,071	0.026	0.066	0.048	0.049	0.044	0.011
8	864	1,168	0.018	0.044	0.142	0.124	0.069	0.080

Figure 2.3: `Julia` performance on various inputs. All running times are is seconds. The #
columns specify the number of vertices in each input. The single missing running time is for
a case where the program ran out of memory. The quality of approximation is in the title
of the column. The VER column is for running the VE-Fréchet (retractable) algorithm on
the original input curves, which is much slower than the exact algorithm, which computes
the exact Fréchet distance, but uses simplification internally for speed.

Input	P #	Q #	≈4	≈1.1	≈1.01	Exact	VER
1	10,400	11,815	0.071	0.250	5.372	0.593	92.869
2	16,318	14,719	0.091	0.240	1.142	69.525	66.844
3	22,310	4,607	0.087	0.232	1.041	49.426	25.032
4	56,730	91,743	0.307	0.563	1.892	2.390	---
5	6,103	9,593	0.042	0.032	0.175	3.657	19.618
6	2,696	1,467	0.077	0.294	4.913	2.946	283.538
7	1,062	1,065	0.145	0.275	0.455	0.666	382.455
8	858	1,162	0.034	0.100	1.537	0.159	0.709

Figure 2.4: Python performance in seconds on the inputs given in Figure 2.2. The missing
runtime is for a case where the test code ran out of memory.

3. The retractable Fréchet distance

3.1. The retractable path in a directed graph

Let $G = (V, E)$ be a directed graph with n vertices and m edges, with weights $w : E \to \mathbb{R}$ on
the edges (assume for simplicity of exposition that they are all distinct). Consider a simple
path π in G. The ***bottleneck*** of π is $b(\pi) = \max_{e \in \pi} w(e)$. For any two vertices s and t in
G, let $\Pi(s, t)$ denote the set of all simple paths from s to t in G. The *bottleneck distance*
between s and t is $d_B(s, t) = \min_{\pi \in \Pi(s,t)} b(\pi)$. The *unique* edge, (under our assumption of
distinct weights on the edges), that realizes $d_B(s, T)$ is the ***bottleneck edge*** of s and t,
denote by $b(s, t)$.

Literature on computing the bottleneck edge/path. This variant of the problem is a min max, which is equivalent to the max min (maximize cheapest edge on the path) via negation of the prices of edges. If the edges are sorted, then computing the bottleneck edge can be done in linear time. Dijkstra can be modified to solve this problem [CKT+16], in $O(n \log n + m)$ time. Similarly, the problem is readily solvable in linear time if the underlying graph is a DAG. Gabow and Tarjan [GT88] showed an algorithm with running time $O(m \log^* m)$, and some minor further improvements are known [CKT+16].

A path that is bottleneck-optimal for subpaths. Here, we study the following harder variant of the bottleneck problem.

Definition 3.1. For two vertices s, t of a directed graph G with distinct weights on its edges, the ***retractable*** path from s to t, is the unique path that contains the edge $\mathcal{b}(s, t) = u \to v$, and furthermore, the subpath from s to u, and from v to t are both retractable.

Importantly, we are interested in computing the retractable path itself (not just the bottleneck). Confusingly, and this is not immediate to see, the standard dynamic programming algorithm for DAGs does not work in this case.

Modifying Dijkstra. It is not hard to see that a minor simplification to Dijkstra's algorithm, started from s, leads to a $O(n \log n + m)$ time algorithm for computing the retractable tree, containing the retractable paths from s to all the vertices in G (we assume here that all of G is reachable from s). Indeed, if C is in the current set of vertices already visited, the algorithm always handles the next cheapest edge in the directed cut $(C, V \setminus C)$. As in Dijkstra's algorithm, one can maintain a variable $d(v)$, to store the weight of cheapest edge from a vertex already visited to the (yet unvisited) vertex v. Setting $d(v) = \min(d(v), \mathbf{w}(u \to v))$ when handling the edge $u \to v$, records this information. Under this vertex based accounting, the algorithm always next handles the vertex with minimum d value that is not yet visited (and these values can be maintained in a heap). Gabow and Tarjan [GT88] described how to modify Dijkstra to solve the bottleneck problem – what we point in the next lemma is that it is somewhat stronger – it computes the retractable path.

Lemma 3.2. *The above algorithm computes a directed tree T rooted at s, such that for any vertex u, the path in T from s to u is retractable. The running time of the algorithm is $O(n \log n + m)$.*

Proof: The running time follows as the algorithm performs $O(m)$ decrease-keys and $O(n)$ delete-min operations, which take $O(1)$ and $O(\log n)$ time respectively if using a Fibonacci heap.

As for correctness – we analyze a somewhat slower implementation of the algorithm (i.e., this is a variant of Prim's algorithm using cuts). The algorithm repeatedly handles the cheapest edge not handled yet that is in the directed cut $(C, \overline{C})^3$, where C is the set of

---

[3] $(C, \overline{C}) = \{u \to v \in E(G) \mid u \in C, v \in V(G) \setminus C\}$.
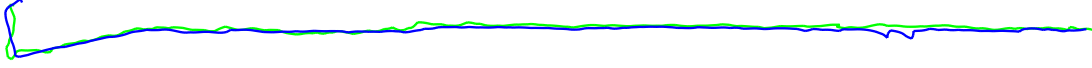
Figure 3.1: Close encounters of the Fréchet type: An example of two curves (from Geolife GPS tracks) that are made out of 547 vertices, such that their retractable discrete Fréchet distance was computed by visiting only $1,144$ edges, while the whole diagram has $74,236$ distinct cells. For more details, follow this link. Informally, for examples where the Fréchet distance is dramatically smaller than the diameter of the curves, the retractable Fréchet distance (discrete or continuous) is computed by the new algorithm in near linear time.

vertices visited so far. Initially, $C = \{s\}$ and the min-heap is initialized to hold the set $(C, \overline{C})$. The algorithm now repeatedly extracts the minimum edge $u \to v$ in the heap. If $v \in C$, then the algorithm continues to the next iteration. Otherwise, the algorithm marks the v as visited, and adds all the outgoing edges from v to the heap. It is easy to verify (by induction) that this slower algorithm, and the algorithm described above compute the same bottleneck-tree.

As for the correctness of the slower algorithm – the proof is by induction on the number of edges of the retractable path. Consider the case that the retractable path is of length one – namely, the edge $e = s \to u$. Let $w = w(e)$. Clearly, before the algorithm handles e, all the weights being handled are strictly smaller than e, and thus the algorithm can not arrive to u. Since e is scheduled by the algorithm, the claim readily follows.

So, consider a retractable path π from s to u that contains more than one edge, and let $e = x \to y$ be its bottleneck edge. Let $w = w(e)$. Observe that the algorithm would handle all the edges of π before handling any edge with strictly larger weight than w. As such, for our purposes, assume that e is the heaviest edge in the graph. Let S be all the vertices visited just before e was handled, and let $T = V \setminus S$. Consider the two induced graphs G_S and G_T, and observe that one can apply induction to both parts, separately, as there are no edges except e between the two parts (there might be edges going "back", but these can be ignored, as the vertices of S were already handled by the algorithm). It thus follows that the algorithm computed π as the desired path. ∎

3.2. The retractable discrete Fréchet distance

3.2.1. The discrete Fréchet distance

Let $\pi = p_1, \ldots, p_n$ and $\sigma = q_1, \ldots, q_m$ be two sequences of points in some normed space. Conceptually, we have two agents starting at p_1 and q_1 respectively. During an atomic time interval, one of them can jump forward to the next point in the sequence (one can allow both to jump forward in the same time, but we disallow it for the sake of simplicity of exposition). In the end, both agents have to arrive to p_n and q_m, respectively, and our purpose is to minimize the maximum distance between them during this motion.

Definition 3.3. Consider two sequences $\pi = p_1, \ldots, p_n$ and $\sigma = q_1, \ldots, q_m$, a ***discrete morphing*** is an x/y-monotone path $\tau = (1,1), \ldots, (n,m)$ in the grid graph defined over the set

18

of points $U = [\![n]\!] \times [\![m]\!]$ from $(1,1)$ to (n,m). For a vertex $(i,j) \in U$, let $\hbar(i,j) = \|p_i q_j\|$ be its associated *weight*. The path τ is restricted to vertical and horizontal edges of the grid. The *width* of τ is $\omega_{\pi,\sigma}(\tau) = \max_{(i,j)\in V(\tau)} \hbar(i,j)$. The ***discrete Fréchet distance*** between π and σ is the minimum of width of any morphing between the two sequences.

3.2.2. The retractable discrete Fréchet distance

Definition 3.4. Let $\tau = z_1, \ldots, z_{n+m-1} \in U$ be a discrete morphing between two sequences $\pi = p_1, \ldots, p_n$ and $\sigma = q_1, \ldots, q_m$. Let $D(i,j) = \max_{k:i<k<j} \hbar(z_k)$ be the inner width of the morphing $\tau(i,j) = z_i, z_{i+1}, \ldots, z_j$. For simplicity assume all pairwise distances $\|p_i q_j\|$ are distinct. The unique point $\textit{\&}(i,j)$ realizing the minimum $D(i,j)$, overall possible grid monotone paths τ' between z_i and z_j, is the *bottleneck* between i and j. The ***retractable discrete Fréchet morphing*** between π and σ is the unique morphing τ, such that
  (i) $z_1 = (1,1)$, $z_{n+m-1} = (n,m)$, and
  (ii) for all $i < j$, we have $\textit{\&}(i,j) \in \tau(i,j)$.

Although the weights here are defined on the vertices, it is easy enough to interpret them as being on the edges (by for example, assigning a grid edge $z \to z'$ the weight $\max(\hbar(z), \hbar(z'))$). Plugging this (implicit) graph into Lemma 3.2 readily yields the following result.

Lemma 3.5. *Given two sequences $\pi = p_1, \ldots, p_n$ and $\sigma = q_1, \ldots, q_m$, the retractable discrete Fréchet morphing between π and σ can be computed in $O(nm \log(nm))$ time.*
  More generally, if the Fréchet distance between π and σ is ℓ, and

$$\tau = |\{(p,q) \mid p \in \pi, q \in \sigma, \|pq\| \leq \ell\}|,$$

then the running time of the algorithm is $O(\tau \log \tau)$.

Remark 3.6. (A) The first running time bound of Lemma 3.5 is a worst case bound, and for many realistic inputs it is much smaller if the (discrete) Fréchet distance between them is small compared to the diameter of the two curves. See Figure 3.1 for an example.
 (B) Note that if one has to explore a large fraction of the grid (i.e., $\tau = \Omega(nm)$), then the retractable Fréchet algorithm is slower than the standard discrete Fréchet algorithm, because of the use of a heap.

4. The **VE**-Fréchet distance

In this section, we give a formal definition of the VE-Fréchet distance, show some of its basic properties in relation to other variants of the Fréchet distance, and briefly discuss practical considerations.

4.1. Definition and basic algorithm

The input is two polygonal curves $\pi = p_1 p_2 \cdots p_n$ and $\sigma = q_1 q_2 \cdots q_m$. This induces the free space diagram $R = R(\pi, \sigma)$ (Definition 1.2), which is a rectangle R, partitioned into a non-uniform $(n-1) \times (m-1)$ grid, where the ith edge $p_i p_{i+1} \in \pi$ and the jth edge $q_j q_{j+1} \in \sigma$, induces the (i,j) grid cell $C_{i,j}$.

Definition 4.1. For two points $u, v \in \mathbb{R}^d$, let $\overrightarrow{uv} = \frac{v-u}{\|vy\|}$ be the (unit length) **direction** vector from u to v.

Let $\boldsymbol{p}_i = \overrightarrow{p_i p_{i+1}}$ and $\boldsymbol{q}_j = \overrightarrow{q_j q_{j+1}}$. Let $x_i = \|\pi[p_1, p_i]\|$ and $y_j = \|\sigma[q_1, q_j]\|$. The grid cell $C_{i,j}$ is the subrectangle $[x_i, x_i + \|p_i p_{i+1}\|] \times [y_j, y_j + \|q_j q_{j+1}\|]$. The elevation function for any point (x, y) inside $C_{i,j}$ is given by

$$e(x,y) = \|\pi(x) - \sigma(y)\| = \left\| p_i + (x - x_i) * \boldsymbol{p}_i - q_j - (y - y_j) * \boldsymbol{q}_j \right\|.$$

The function $e(x,y)$ is a smooth convex function.[4] More pertinent for our purposes, is that the minimum of the elevation function along each boundary edge of $C_{i,j}$ is unique, and corresponds to the distance of a vertex of one curve (say $p_i \in \pi$) to the closest point on an edge (i.e., $q_j q_{j+1}$) of the other curve. This minimum along an edge of $C_{i,j}$ is a **portal**. The portals on the left and bottom edges of $C_{i,j}$ are the *incoming portals*, and the other two are *outgoing portals*.

The standard Fréchet distance asks for computing the x/y-monotone path between the opposite corners of the free space, while minimizing the maximum height along this path. The corresponding Fréchet morphing has the property that its intersection with a cell is either a segment (or empty). We restrict the Fréchet morphing further, by requiring that inside a cell, the morphing must be a straight segment connecting two portals of the cell. As a result, we have to give up on the monotonicity requirement (more on that shortly).

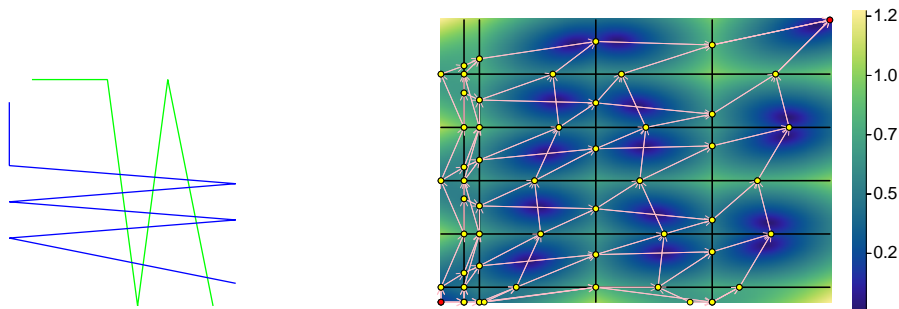Importantly, computing the new VE-Fréchet distance is now a graph search problem, in the following graph.

Figure 4.1: Left: Two curves. Right: Their elevation function, and the associated graph. See here for more info.

---

[4]Specifically, it is a square root of a paraboloid, as a tedious but straightforward calculation shows.

Definition 4.2. Given two curves $\pi = p_1 p_2 \cdots p_n$ and $\sigma = q_1 q_2 \cdots q_m$, their **VE graph** $G_{\mathsf{VE}} = G_{\mathsf{VE}}(\pi, \sigma)$ is a DAG defined over the grid of the free space diagram $R = R(\pi, \sigma)$, where every grid cell has four vertices on its boundary (some of them might coincide), corresponding to the portals on the edges. Here, every cell has edges from each of its incoming portals to each of its outgoing portals (i.e., four edges in total when the portals do not coincide). In addition, there are two special vertices – the start vertex s (i.e., bottom left corner of R), and target vertex t (i.e., top right corner of R). The portals on the grid edges adjacent to s and t are moved to s and t, respectively. Every vertex v of this graph, has an associated location $(x, y) \in R$, which in turn corresponds to two points $p_v = \pi(x)$ and $q_v = \sigma(y)$. The **height** of v is the elevation of (x, y) – that is $\|p_v q_v\|$.

For an example of the G_{VE} graph, see Figure 1.3. The VE-Fréchet distance between π and σ is simply the bottleneck distance between s and t in G_{VE}. We might as well compute the retractable version.

Definition 4.3. Given two polygonal curves π and σ, the **retractable VE-Fréchet morphing** is the unique morphing m realized by the retractable path in $G_{\mathsf{VE}}(\pi, \sigma)$ from s to t, see Definition 3.1. The **VE-Fréchet distance** is the maximum elevation of any point along m, denoted by $\mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma)$.

Putting the above together, and using the algorithm of Lemma 3.2, we get the following.

Theorem 4.4. *Given two polygonal curves π and σ with n and m vertices respectively, their VE-Fréchet distance, and the corresponding retractable VE Fréchet morphing $m_{\mathsf{VE}}(\pi, \sigma)$ that realizes it, can be computed in $O(nm \log(nm))$ time.*

We emphasize that that the running time bound in this theorem is pessimistic, and in many cases the algorithm is significantly faster. For an example of the output of the algorithm, see Figure 4.2.

4.1.1. Reachable complexity

Definition 4.5. For two curves π and σ, consider the VE graph $G_{\mathsf{VE}} = G_{\mathsf{VE}}(\pi, \sigma)$. For a threshold $r \geq 0$, let $\mathsf{N}_{\leq r}(\pi, \sigma)$ be the number of vertices of G_{VE} that are reachable from the start vertex via paths with maximum elevation at most r. The quantity $\mathsf{N}_{\leq r}(\pi, \sigma)$ is the **r-reachable complexity** of π and σ.

A similar notion (relative free space complexity) was used by Driemel *et al.* [DHW12]. If $r = \mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma)$, then the algorithm for computing the VE morphing only explores vertices in the r-reachable region, which readily implies the following.

Corollary 4.6. *Given two polygonal curves π and σ with n and m vertices respectively, their VE-Fréchet distance, and the corresponding retractable VE Fréchet morphing $m_{\mathsf{VE}}(\pi, \sigma)$ that realizes it, can be computed in $O(n + m + \mathsf{N} \log \mathsf{N})$ time, where $\mathsf{N} = \mathsf{N}_{\leq r}(\pi, \sigma)$ and $r = \mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma)$.*
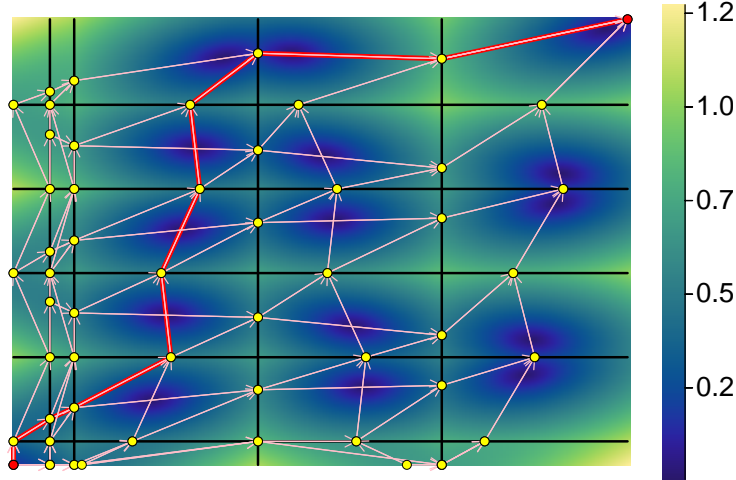
Figure 4.2: The VE-Fréchet morphing for the two curves from Figure 1.3. Note, that the solution is "slightly" not x/y-monotone. For animation of this morphing, follow the link.

Remark 4.7. The above implies that the worst case, for the above algorithm, is when $N = nm$. This happens for example (somewhat counterintuitively) when π and σ are far from each other, and their VE-Fréchet distance is realized by the distance of their endpoints (say, in the start of the morphing). This somewhat absurd situation can happen for real world inputs – follow this link for an example. Furthermore, this quadratic behavior is probably unavoidable, as computing the VE-Fréchet distance probably requires quadratic time in the worst case. We offer some approaches to avoid this for real world input later on.

Fortunately, for many inputs where the two curves are close to each other (e.g., their VE-Fréchet distance is significantly smaller than their diameters), the running time of the algorithm of Corollary 4.6 is $O\big((n + m)\log(n + m)\big)$, see Figure 3.1 for an example.

4.2. Basic properties of the **VE** Fréchet morphing/distance

Lemma 4.8. *For any two curves π and σ, we have that $\mathsf{d}_{\mathcal{F}}(\pi, \sigma) \geq \mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma)$.*

Proof: Consider the Fréchet morphing m realizing the Fréchet distance $\alpha = \mathsf{d}_{\mathcal{F}}(\pi, \sigma)$. Track the path m in the free space diagram, and inside each cell snap it to the portals of the corresponding edges it crosses. Clearly, this results in a valid morphing that is a valid path in G_{VE} from the source to the target, and its bottleneck value is no bigger than the original value (as by definition portals were the minimum elevation points along the corresponding edges). However, $\beta = \mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma)$ being the minimum bottleneck path in this graph from the start vertex to the target vertex, is definitely not bigger than α. ∎

Observation 4.9. *The VE-Fréchet morphing is monotone over the vertices.*

Specifically, let m be the VE-Fréchet morphing between π and σ. If we project m on the x-axis, denoted by $x(m)$, we get a path from 0 to $\|\pi\|$ on the real line. Note, that this path is

22

not necessarily monotone, but importantly, for any vertex $v \in \pi$ it is monotone. Specifically, let α be the distance of v along π from its start. Then, the path $x(m)$ crosses from one side of α to the other side, exactly once (assuming v is an internal vertex). To see why this is true, observe any morphing must cross the vertical line $x = \alpha$. This line has vertices along it, and observe that in $G_{VE}(\pi, \sigma)$ it is only possible to cross this line by using one of these vertices, as no edge crosses from one side of this line to the other (i.e. the vertices on this line are a separator). However, vertices along this line have outgoing edges only to vertices that are to the right (or remain on the same x, but these target vertices are on grid edges adjacent on to the right to this line).

Definition 4.10. Given a polygonal curve $\pi = p_1 p_2 \cdots p_n$, a ***refinement*** of π is a polygonal curve $\pi' = p_1 Q_1 p_2 Q_2 p_3 \cdots Q_{n-1} p_n$, where for all $1 \le i < n$, Q_i is a (possibly empty) sequence of points occurring in order along the (directed) edge $p_i p_{i+1}$.

Observation 4.11. *If π', σ' are refinements of π, σ respectively, then $\mathsf{d}_{\mathcal{F}}(\pi', \sigma') = \mathsf{d}_{\mathcal{F}}(\pi, \sigma)$ – indeed, the additional vertices from a refinement do not change the underlying curve.*

However, $\mathsf{d}_{\mathcal{F}}^{ve}(\pi', \sigma') \ge \mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma)$ – indeed, refinement corresponds to adding vertical and horizontal lines in the free space diagram that the morphing can cross only once (i.e., the morphing is monotone at the vertices). As such, the morphing realizing $\mathsf{d}_{\mathcal{F}}^{ve}(\pi', \sigma')$ is determined by more constraints, and it is thus (potentially) larger.

5. Computing the regular Fréchet distance

5.1. Direct Monotonization

The basic idea to compute the Fréchet distance using the new algorithm is to refine the curves till their VE-Fréchet morphing becomes monotone.

Observation 5.1. *If the VE-Fréchet morphing is x/y-monotone, then it realizes the (regular) Fréchet distance between the curves. Indeed, if the morphing is monotone then it is a valid morphing considered by Definition 1.4 for the Fréchet distance, and thus $\mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma) \ge \mathsf{d}_{\mathcal{F}}(\pi, \sigma) \ge \mathsf{d}_{\mathcal{F}}^{ve}(\pi, \sigma)$, where the latter inequality is from Lemma 4.8.*

5.1.1. How morphing encodes the motion.

A VE-Fréchet morphing m in a free space diagram $R = R(\pi, \sigma)$ of two curves π and σ with n and m vertices, respectively, is a polygonal path with $t = n + m + O(1)$ vertices. Thus, a morphing can be represented as $m = (x_1, y_1), \ldots, (x_t, y_t)$. A consecutive pair of vertices of the morphing $(x_i, y_i), (x_{i+1}, y_{i+1})$ corresponds to two directed subsegments $\tau \subseteq \pi$ and $\nu \subseteq \sigma$. The matching of these two subsegments encodes a linear motion starting at the start points, and ending and the endpoints. See Figure 5.1.

Definition 5.2. Given a morphing $m = (x_1, y_1), \ldots, (x_t, y_t)$, let $X_i = \max_{j=1}^{i} x_j$ and $Y_i = \max_{j=1}^{i} y_j$, for $i = 1, \ldots, t$. Let $\mathrm{mono}(m) = (X_1, Y_1), (X_2, Y_2), \ldots, (X_t, Y_t)$ denote the ***direct monotonization*** of m (which can be computed in $O(t)$ time).

Figure 5.1: Left: Two curves. Right: Their Fréchet morphing. Since the Fréchet morphing computed inside each cell of the free space diagram is a segment, it is enough to mark all the critical configurations when the morphing enters/leaves a cell. Each critical configuration is depicted on the right by a segment connecting the two points being matched. In between two such configurations, the morphing is the linear interpolation between the two configurations. See here for an animation of this specific morphing.

Observation 5.3. *Let m_{VE} be the VE-Fréchet morphing between π and σ. Then $\mathsf{d}_{\mathcal{F}}(\pi, \sigma) \in [\mathsf{d}^{ve}_{\mathcal{F}}(\pi, \sigma), \omega(\mathrm{mono}(m_{VE}))]$, where the lower bound follows from Lemma 4.8 and the upper bound follows as $\mathrm{mono}(m_{VE})$ is a monotone morphing and thus is a valid morphing considered by Definition 1.4 for the Fréchet distance.*

For many natural inputs $\omega(m_{VE}) = \omega(\mathrm{mono}(m_{VE}))$. Specifically, this occurs if either m_{VE} is monotone to begin with or if the "zigzagging" in m_{VE} is happening in parts where the leash is relatively short, and thus the fixup due to direct monotonization does not affect the global bottleneck. Note that in this case, $\mathsf{d}^{ve}_{\mathcal{F}}(\pi, \sigma) = \omega(\mathrm{mono}(m_{VE}))$, and so by Observation 5.3, $\mathrm{mono}(m_{VE})$ is an optimal morphing realizing $\mathsf{d}_{\mathcal{F}}(\pi, \sigma)$.

Given a morphing m, one can compute its width in linear time, as by convexity it is realized by the elevation of one of the vertices of m. Thus, a first step to trying to compute $\mathsf{d}_{\mathcal{F}}(\pi, \sigma)$, is to compute the VE-Fréchet morphing m_{VE} between π and σ, and then directly monotonize it. If $\omega(m_{VE}) = \omega(\mathrm{mono}(m_{VE}))$, then we have computed the Fréchet distance and so we are done. Otherwise, we will use refinement to narrow the interval $[\mathsf{d}^{ve}_{\mathcal{F}}(\pi, \sigma), \omega(\mathrm{mono}(m_{VE}))]$, as described below.

5.1.2. Monotonization via refinement

The lower/upper bound. Let m_{VE} be the VE-Fréchet morphing between π and σ. By Observation 5.3, $\mathsf{d}_{\mathcal{F}}(\pi, \sigma) \in [\mathsf{d}^{ve}_{\mathcal{F}}(\pi, \sigma), \omega(\mathrm{mono}(m_{VE}))]$. Thus we have initial upper and lower bounds for $\mathsf{d}_{\mathcal{F}}(\pi, \sigma)$, and our approach is to repeatedly improve these bounds until they are equal, at which point the computed morphing realizes the Fréchet distance between the two curves.

Observation 5.4. *Let $\varepsilon \in (0, 1)$. One can refine π and σ into curves π' and σ', with VE-Fréchet morphing m'_{VE}, such that*

$$\omega(\mathrm{mono}(m'_{VE})) - \mathsf{d}^{ve}_{\mathcal{F}}(\pi', \sigma') \leq \varepsilon$$

Indeed, create π' and σ' by introducing vertices along π and σ respectively, such that no edge has length exceeding $\varepsilon/2$. Recall that by Observation 4.9, m'_{VE} is already monotone over

24

vertices, and thus mono($m'_{\textsf{VE}}$) *does not change the edges from the curves that points in the morphing are mapped to. Thus the direct monotonizations of the* VE *morphings of* π' *and* σ' *can increase the distance by at most twice the maximum edge length.*

Thus, in the limit, one can compute the Fréchet distance from a refinement of the two curves. Fortunately, there is a finite refinement that realizes the Fréchet distance.

Lemma 5.5. *Let* π *and* σ *be two polygonal curves of total complexity* n. *Then, there is a set* S *of* $O(n^2)$ *vertices, such that if we refine* π *and* σ *using* S, *then for the resulting two curves,* π' *and* σ', *we have that* $\mathsf{d}^{ve}_{\mathcal{F}}(\pi', \sigma') \leq \mathsf{d}_{\mathcal{F}}(\pi, \sigma) = \omega(\mathrm{mono}(m_{\textsf{VE}}(\pi', \sigma')))$.

Proof: Let $r = \mathsf{d}_{\mathcal{F}}(\pi, \sigma)$, and introduce a vertex in the interior of an edge of π if this interior point is in distance exactly r from some vertex of σ, and vice versa. Let π' and σ' be the resulting refined curves. Consider the grid H of $R(\pi', \sigma')$, and any edge e of this grid. The value of the elevation function on e is an interval: $e(e) = \{e(p) \mid p \in e\}$, and this interval cannot contain r in its interior (if it did, we would have broken it by introducing a vertex at the corresponding interior point). Thus any edge in H whose elevation interval has a value above r, cannot contain any interior point with value below r, and thus such an edge is infeasible (in its interior) for the VE-Fréchet morphing $m_{\textsf{VE}}(\pi', \sigma')$, as we know $\mathsf{d}^{ve}_{\mathcal{F}}(\pi', \sigma') \leq r$. Namely, every edge of the free space is either completely feasible (for r), and or completely infeasible.
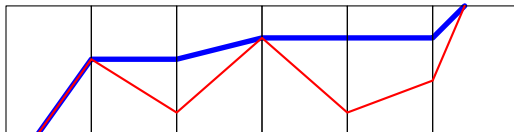

Figure 5.2: The red path has a zigzag in this row. The blue path depicts the monotonized path.

Now, if $m = m_{\textsf{VE}}(\pi', \sigma')$ is monotone then we are done. Otherwise, the portions of m that are not monotone can be broken into a sequence of "zigzags" in columns and rows. Consider such a (say) row with non-monotonicity. The path (i.e., morphing) enters the row from the bottom on the left side, and leaves on the right from the top, see Figure 5.2. The monotonized portion of this path inside this row enters and leaves through the same extreme portals, but importantly, it uses the same edges. All these edges have maximum elevation at most r, and thus the width of the monotonized morphing is at most r (and thus, equal to r), establishing the claim. ∎

Remark 5.6. It is straightforward to check if a given morphing m is monotone. If it is not, then compute for each curve the portions where this morphing travels on them backward, and introduce a vertex in the middle of each backwards portion. Now, we recompute a morphing on the refined curve, Clearly this reduces the "back and forth" non-monotone motion on the two curves. One can naturally repeat this process several times till the non-monotonicity disappears, or becomes so small that it can be ignored. We refer to this as **bisection refinement**.

5.2. Computing the regular Fréchet distance

The above suggests natural strategies for computing the Fréchet distance:

(I) First, compute the VE-Fréchet morphing. We now apply the bisection refinement, described in Remark 5.6, to the resulting morphing (recomputing the morphing using VE-Fréchet if needed). In practice, a few rounds of this bisection refinement makes these portions of the morphing involve leashes shorter than the bottleneck (even after direct monotonization), and we end up with the Fréchet distance.

Remark 5.7. Underlying the bisection refinement strategy is the implicit assumption that the matching induced by the VE-Fréchet morphing is (relatively) ***stable***, and matches roughly the same portions of the curves after refinement. This seems to be the case for all the curves we considered. Understanding this stability might be an interesting venue for future research.

(II) Another (more systematic) approach to refinement is to consider all the vertices V (say) of π involved in a subpath of the morphing that is not monotone on an edge e of σ. We refine e along all the points that are intersections of bisectors of pair of points of V with e. Since repeated application of this process converges to the refinement set of Lemma 5.5 (the set used in this lemma is too expensive to compute explicitly), this implies that sooner or later this yields the Fréchet distance (after direct monotonization). Essentially, this performs a search of the monotonicity events, see Section 1.2.2, that are relevant to the current morphing.

5.3. Speed-up via simplification

A natural approach to try and avoid the quadratic complexity of computing the VE-Fréchet distance is to use simplification of the input before computing the distance. Here we explore how to use simplification, so that we can compute the exact distance. We start with a simple linear-time simplification algorithm [AHK+06, DHW12].

Algorithm 5.8. For a polygonal curve $\pi = p_1, p_2, \ldots, p_n$, and a parameter $\delta > 0$, the algorithm marks the initial vertex p_1, and sets it as the current vertex. The algorithm repeatedly scans π from the current vertex till reaching the first vertex p_i that is in distance at least δ from the current vertex. It marks p_i, and sets it as the current vertex. The algorithm repeats this until reaching the final vertex, which is also marked. The resulting subcurve connecting the marked vertices $\overline{\pi} = \mathrm{simpl}(\pi, \delta)$ is the ***δ-simplification*** of π, and it is not hard to verify (and it is also well known [DHW12]) that $\mathsf{d}_{\mathcal{F}}(\pi, \overline{\pi}) \leq \delta$.

Observe that $\mathsf{d}_{\mathcal{F}}(\pi, \sigma) - 2\delta \leq \mathsf{d}_{\mathcal{F}}(\overline{\pi}, \overline{\sigma}) \leq \mathsf{d}_{\mathcal{F}}(\pi, \sigma) + 2\delta$. (same holds for the VE version), where $\overline{\pi} = \mathrm{simpl}(\pi, \delta)$ and $\overline{\sigma} = \mathrm{simpl}(\sigma, \delta)$.

Lemma 5.9. *Given a curve π with n vertices, and a simplified curve $\overline{\pi}$ computed by applying Algorithm 5.8 to π, with a parameter δ, one can compute, in $O(n)$ time, a Fréchet morphing between π and $\overline{\pi}$ of width $\ell \leq 2\delta$, such that $\mathsf{d}_{\mathcal{F}}(\pi, \overline{\pi}) \leq \ell$.*

Proof: Consider the case that $\overline{\pi}$ is a single segment s. Consider first the "silly" morphing m', that matches every vertex of π to its nearest neighbor on s (in between, it interpolates linearly). Clearly, this is a weak morphing between π and $\overline{\pi}$, but it is also clearly optimal. Consider the monotone version m of m' that one gets from m' by never moving back along s, and let ℓ be its width. It is not hard to verify that $\ell \leq 2\delta$.

Now remove the assumption that $\overline{\pi}$ is a single segment. By construction, $\overline{\pi}$ is a subsequence of π, including the start and end vertices. Thus the edges of $\overline{\pi}$ partition π into pieces, which start and end at the corresponding end points of the edge. Thus in this more general case, we can simply apply the above argument to each edge of $\overline{\pi}$ and its corresponding piece of π, and then concatenate all these morphings together. ∎

Given a δ-simplification $\overline{\pi}$, the above gives us a fast way to compute a morphing between π and $\overline{\pi}$. This morphing may not be optimal, but its error is bounded by 2δ, and we are free to set δ to our desired value (depending on which the simplified curve may have significantly fewer vertices). This basic approach works for any subcurve specified by a subsequence of the vertices of π.

Remark 5.10. A natural conjecture is that for any sub-curve $\overline{\pi}$ of π defined by a subset of the vertices of π (including the same start and end vertices), the above quantity $\mathsf{d}_{\mathcal{F}}(\pi, \overline{\pi}) \leq \ell \leq 2\mathsf{d}_{\mathcal{F}}(\pi, \overline{\pi})$. To see that this is false, let $m > 1$ be an integer, and let $\delta = 1/2m$, and consider the one dimensional curve and its subcurve:

$$\pi = 0, 1, \delta, 1 - \delta, 2\delta, 1 - 2\delta, \ldots, 1/2 - \delta, 1/2 + \delta, 1/2.$$
$$\overline{\pi} = 0, \quad 1 - \delta, 2\delta, 1 - 2\delta, \ldots, 1/2 - \delta, 1/2 + \delta, 1/2.$$

The above algorithm would compute $\ell \approx 1$ (or $\ell \approx 1/2$ if it computes the exact Fréchet distance between every segment of the simplification and its corresponding subcurve), but $\mathsf{d}_{\mathcal{F}}(\pi, \overline{\pi}) = O(\delta)$.

5.3.1. Combining morphings

Getting a monotone morphing via simplification. The input is the two curves π and σ, and a parameter δ. We compute $\overline{\pi} = \text{simpl}(\pi, \delta)$ and $\overline{\sigma} = \text{simpl}(\sigma, \delta)$. Next, compute

$$m_1 = \text{mono}(m_{\mathsf{VE}}(\pi, \overline{\pi})), m_2 = \text{mono}(m_{\mathsf{VE}}(\overline{\pi}, \overline{\sigma})), \quad \text{and} \quad m_3 = \text{mono}(m_{\mathsf{VE}}(\overline{\sigma}, \sigma)).$$

For the sake of simplicity of exposition, think about these morphings m_i as piecewise linear 1-to-1 functions. For example, $m_1 : [0, \|\pi\|] \to [0, \|\overline{\pi}\|]$. Thus, combining the above three morphings $m = m_3 \circ m_2 \circ m_1 : [0, \|\pi\|] \to [0, \|\sigma\|]$ yields the desired monotone morphing.

Remark 5.11. In practice, one does not need to compute $m_1 = \text{mono}(m_{\mathsf{VE}}(\pi, \overline{\pi}))$, but one can do a "cheapskate" greedy morphing (which is monotone) as done by the algorithm of Lemma 5.9.

The algorithm for combining two morphings is a linear time algorithm somewhat similar in nature to the merge subroutine used in merge-sort. We omit the straightforward but tedious details.

Lemma 5.12. *Given two compatible x/y-monotone morphings m_1 and m_2 (i.e., polygonal curves in $\mathbb{R}^2$), their combined morphing $m_2 \circ m_1$ can be computed in linear time.*

Remark 5.13. The above provides us with a "fast" algorithm for computing a morphing between two input curves π and σ in time that is effectively near linear (if the simplification is sufficiently aggressive). Specifically, we simplify the two curves, and then compute the monotone Fréchet distance between the simplified curves by computing the VE-Fréchet morphing and refining it till it becomes monotone. We then stitch the three morphings together to get a morphing for the original two curves. Clearly, by making the simplification sufficiently small, this converges to the optimal Fréchet morphing.

The question is thus the following: Given a Fréchet morphing, can we test "quickly" whether it realizes the Fréchet distance between the two curves (without computing the Fréchet distance directly).

5.4. Computing the exact Fréchet distance using simplification

5.4.1. Morphing sensitive simplification

The challenge in using simplification for computing the Fréchet distance is to be able to argue that the computed morphing is optimal. To this end, we are interested in computing a lower bound on the Fréchet distance from simplified curves, such that the upper/lower bounds computed match, thus implying that the computed solution is indeed optimal.

A weak lower bound. Consider two curves π and σ, and their respective simplifications $\overline{\pi}$ and $\overline{\sigma}$. By the triangle inequality, we have

$$\mathsf{d}_{\mathcal{F}}(\pi, \sigma) \geq \ell(\pi, \sigma) = \mathsf{d}_{\mathcal{F}}(\overline{\pi}, \overline{\sigma}) - \mathsf{d}_{\mathcal{F}}(\pi, \overline{\pi}) - \mathsf{d}_{\mathcal{F}}(\sigma, \overline{\sigma}).$$

The ***lower bound*** ℓ is usually easy to compute (or bound from below), since the simplifications usually provide an immediate upper bound on $\mathsf{d}_{\mathcal{F}}(\pi, \overline{\pi})$ and $\mathsf{d}_{\mathcal{F}}(\sigma, \overline{\sigma})$.

Definition 5.14. For any $\psi > 1$, a Fréchet morphing m of π and σ is a ***ψ-approximate morphing*** if $\mathsf{d}_{\mathcal{F}}(\pi, \sigma) \leq \omega(m) \leq \psi \mathsf{d}_{\mathcal{F}}(\pi, \sigma)$. Thus $\ell = \omega(m)/\psi$ is a lower bound on the Fréchet distance between π and σ.

Definition 5.15. For a morphing $m \in \mathcal{M}_{\pi,\sigma}$, a lower bound $\ell \leq \mathsf{d}_{\mathcal{F}}(\pi, \sigma)$, and a point $p = (x, y) \in m$, the ***slack*** at p is $\Delta(p) = \max(\ell - e(p), 0)$. A point $p \in m$ is ***tight*** if $\Delta(p) = 0$.

Consider an optimal (say retractable) Fréchet morphing m between π and σ. If the lowerbound is the Fréchet distance, then we expect only the point in the morphing realizing the bottleneck to be tight. One can map the slack from the morphing back to the original

curves. Specifically, for $v \in \pi \cup \sigma$, let $\max_m(v)$ be the maximum length leash attached to v during the morphing m (this can correspond to the maximum elevation along a vertical or horizontal segment in m).

Definition 5.16. For a vertex $v \in \pi \cup \sigma$, the ***slack*** of v is $\Delta(v) = \max(\ell - \max_m(v), 0)$.

Intuitively, the slack of a vertex is how much one can move it around without introducing too much error – observe that for some vertices the slack is zero, implying they cannot be moved. So, consider a curve $\pi = p_1, \ldots, p_n$, and a "simplified" subcurve of it $\overline{\pi}$. For a vertex $p_i \in \pi$, its ***representative*** in $\overline{\pi}$, denoted by $\text{rep}(p_i)$, is the point p_j, such that p_j is a vertex of $\overline{\pi}$, j is maximal, and $j \leq i$.

Definition 5.17. For some real number $\tau \geq 1$, the curve $\overline{\pi}$ is a (m, τ)-*sensitive simplification* (or simply ***m-simplification***) of π if, for all i, we have that $\|p_i - \text{rep}(p_i)\| \leq \Delta(p_i)/\tau$.

5.4.2. Computing a lower-bound on the Fréchet distance

Consider an edge $e = p_i p_j$ of the simplified curve. The ***width*** of the edge, denoted by $w_{\mathcal{F}}(e) = \mathsf{d}_{\mathcal{F}}(p_i p_j, p_i p_{i+1} \ldots p_j)$, and consider the hippodrome

$$\hbar(e) = e \oplus w_{\mathcal{F}}(e) = \left\{ p \in \mathbb{R}^d \,\middle|\, d(p, e) \leq w_{\mathcal{F}}(e) \right\},$$

where $d(p, e)$ is the distance of p from e. More generally, for a point $p \in e \subseteq \overline{\pi}$, we denote by $w_{\mathcal{F}}(p) = w_{\mathcal{F}}(e)$.

Let $m_\pi : [0, \|\pi\|] \to [0, \|\overline{\pi}\|]$, be a (monotone) morphing of π to $\overline{\pi}$ (m_σ is defined similarly for σ and $\overline{\sigma}$). There is a natural simplification of the elevation function $e(x, y)$. Specifically, we define

$$e'(x, y) = \left\| \overline{\pi}(m_\pi(x)) - \overline{\sigma}\big(m_\sigma(y)\big) \right\| - w_{\mathcal{F}}\big(\overline{\pi}(m_\pi(x))\big) - w_{\mathcal{F}}\big(\overline{\sigma}(m_\sigma(y))\big)$$
$$\leq \left\| \overline{\pi}(m_\pi(x)) - \overline{\sigma}\big(m_\sigma(y)\big) \right\| - \left\| \pi(x) - \overline{\pi}(m_\pi(x)) \right\| - \left\| \sigma(y) - \overline{\sigma}(m_\sigma(y)) \right\| \leq e(x, y).$$

Namely, computing the Fréchet distance using e' (instead of e), would provide us with a lower bound on the Fréchet distance between the two curves. Of course, this Fréchet distance computation can be done directly on the corresponding "elevation" function between the two simplified curves $\overline{\pi}$ and $\overline{\sigma}$:

$$\forall (x, y) \in [0, \|\overline{\pi}\|] \times [0, \|\overline{\sigma}\|] \qquad \overline{e}(x, y) = \|\overline{\pi}(x) - \overline{\sigma}(y)\| - w_{\mathcal{F}}\big(\overline{\pi}(x)\big) - w_{\mathcal{F}}\big(\overline{\sigma}(y)\big).$$

Naturally, one can replace $w_{\mathcal{F}}\big(\overline{\pi}(x)\big)$ and $w_{\mathcal{F}}\big(\overline{\sigma}(y)\big)$ by larger quantities, and still get the desired lower bound. Let $\overline{\mathsf{d}_{\mathcal{F}}}(\pi, \sigma)$ denote this lower-bound on the Fréchet distance.

5.4.3. Computing the exact Fréchet distance

The above suggests an algorithm for computing the exact Fréchet distance. Start with a low quality approximate morphing m between π and σ (this can be computed directly by simplifying the two curves). Use this to get a more sensitive approximation to the two curves, and compute the Fréchet distance between the two simplified curves. This yields a morphing between the two original curves (i.e., upper bound), and a matching lower bound on the Fréchet distance between the two curves. If the two are equal, then we are done.

The critical property of this algorithm, is that it never computes directly the exact Fréchet distance between the two original curves, which might be too large to handle in reasonable time. The result is summarized in the following lemma.

Lemma 5.18. *Let π and σ be two curves, and let m be a ψ-approximate morphing between them, for some $\psi > 1$. Let $\tau \geq 1$ be some constant, and consider (m, τ)-sensitive simplifications $\overline{\pi}, \overline{\sigma}$, of π, σ, respectively. Let $m_2 : \overline{\pi} \to \overline{\sigma}$ be the optimal Fréchet morphing between $\overline{\pi}$ and $\overline{\sigma}$, and let $m_1 : \pi \to \overline{\pi}$, and $m_3 : \overline{\sigma} \to \sigma$, be the natural morphings associated with the simplifications. This gives rise to a natural morphing $m' = m_3 \circ m_2 \circ m_1$ from π to σ. If $\omega(m') = \overline{\mathsf{d}_{\mathcal{F}}}(\overline{\pi}, \overline{\sigma})$, then m' realizes the optimal Fréchet distance between π and σ. where $\overline{\mathsf{d}_{\mathcal{F}}}(\overline{\pi}, \overline{\sigma})$ is the Fréchet distance between $\overline{\pi}$ and $\overline{\sigma}$ under the modified elevation function $\overline{e}$.*

It is easy to verify that for τ large enough the two simplified curves are the original curves, and the above would compute the Fréchet distance. Thus, the resulting algorithm is iterative – as long as the above algorithm fails, we double the value of τ, and rerun the algorithm, till success.

5.5. The sweep distance

5.5.1. Definitions

Another measure of distance between curves is the CDTW (Continuous Dynamic Time Warping) distance. This distance has a neat description in our setting – given a morphing m between π and σ, we define the two functions

$$f(x) = \min_{y:(x,y)\in m} e(x,y) \qquad \text{and} \qquad g(y) = \min_{x:(x,y)\in m} e(x,y).$$

In words, $f(x)$ (resp. $g(y)$) is the minimum length leash attached to the point $\pi(x)$ (resp. $\sigma(y)$) during the morphing of m. We define the ***warping cost*** of m to be

$$\mathrm{cost}(m) = \int_{x=0}^{\|\pi\|} f(x)\,\mathrm{d}x + \int_{y=0}^{\|\sigma\|} g(y)\,\mathrm{d}y. \tag{5.1}$$

The ***Continuous Dynamic Time Warping*** (***CDTW***), between the two curves is

$$\mathsf{d}_{\mathsf{CDTW}}(\pi, \sigma) = \min_{m \in \mathcal{M}^+_{\pi,\sigma}} \mathrm{cost}(m).$$

Computing the CDTW between curves is not easy and only partial results are known, see the introduction for details. We point out, however, that our available machinery leads readily to computing an upper bound on the CDTW, and furthermore, this upper bound can be made to converge to the CDTW distance via simple refinements of the two curves. In particular, for a well behaved morphing, the integrals of Eq. (5.1) can be computed exactly.

5.5.2. The sweep distance

The cost of morphing along a segment inside a cell. Given two directed segments $\tau = p_1 p_2$ and $\tau' = q_1 q_2$, the natural linear morphing between them is the one where the two points move in constant speed. Specifically, at time $t \in [0, 1]$, we have the two moving points $p(t) = (1 - t)p_1 + tp_2$ and $q(t) = (1 - t)q_1 + tq_2$. The cost of this morphing boils down to a function

$$f(x) = \sqrt{ax^2 + bx + c},$$

for some constant a, b, c, and integrating f on an interval $[0, \|\tau\|]$. Using the computer algebra system `maxima` [Max23] yields the following indefinite integral (which we subsequently verified is correct).

$$F(x) = \int f(x)\,\mathrm{d}x = \left(\frac{c}{2\sqrt{a}} - \frac{b^2}{8a^{3/2}} \right) \operatorname{asinh} \frac{2ax + b}{\sqrt{4ac - b^2}} + \left(\frac{x}{2} + \frac{b}{4a} \right) f(x).$$

In particular, $\operatorname{price}(\tau) = F(\|\tau\|) - F(0)$ is the price of the CDTW charged to τ. We repeat the same argument, but now the morphing is interpreted being for a point moving on τ'. This would yield a similar indefinite integral $G(\cdot)$, and the price of τ' is $\operatorname{price}(\tau') = G(\|\tau'\|) - G(0)$.

In the free space diagram, the morphing between τ and τ' corresponds to a segment that lies in a cell, and $\operatorname{price}(\tau) + \operatorname{price}(\tau')$ is the price the sweep distance assigns this segment in the free space diagram.

Computing the sweep distance. The above gives us a new pricing of the edges of the VE graph (see Definition 4.2), as every edge is a segment in the free space diagram. We can then use Dijkstra to compute the shortest path in the VE graph. Naturally, the resulting path is not monotone, but monotonicity can be easily achieved by introducing middle points, as described in Remark 5.6. We then recompute the shortest path getting a morphing, and repeat the refinement step if the morphing is not yet monotone. Usually, one round of refinement seems to suffice. We output the computed distance (for the associated monotone morphing). We refer to this quantity, as the ***sweep distance*** between the two original curves, denoted by $\mathsf{d}_{SW}(\pi, \sigma)$.

This readily yields the following.

Lemma 5.19. *The sweep distance between two curves π and σ can be computed by repeatedly running Dijkstra on the appropriately defined* DAG. *The sweep distance computed is an upper bound on the* CDTW *distance between the two curves.*

Remark 5.20. The number of times Dijkstra has to be invoked by the above algorithm for real world inputs seems to be once or twice, as the Sweep Distance tends to be larger for longer curves (since its an integral (of a non-negative function) over the curves).

Given a curve π, a ***splitting*** of π is the curve resulting from introducing a vertex in the middle of each segment of π. Let π^i denote the curve resulting for i iterations of splitting. We then have the following.

Lemma 5.21. $\lim_{i \to \infty} \mathsf{d}_{SW}(\pi^i, \sigma^i) = \mathsf{d}_{CDTW}(\pi, \sigma)$.

Proof: As i increases, a cell in the free space diagram of π^i and σ^i becomes smaller, corresponding to the distance functions between shorter subsegments of the two curves. In particular, the functions becomes closer to being constant on each cell, and the relevant integral, forming the CDTW distance, is better approximated by the shortest path on the VE graph. ∎

5.5.3. Computing a lower bound on the CDTW distance using the sweep distance

It is natural to try and compute a lower-bound on the CDTW distance, so that one can estimate the quality of the upper-bound computed by the above algorithm. To this end, consider the free space diagram $R = R(\pi, \sigma)$ induced by π and σ, see Definition 1.2. Let Ξ be the set of grid cells of R, and consider the "silly" elevation function,

$$\forall (x, y) \in R \qquad e(x, y) = \min_{(x', y') \in \Xi(x, y)} e(x', y'),$$

where $\Xi(x, y)$ denotes the cell in the grid of R containing the point (x, y). Namely, we flatten the elevation function inside each grid cell, to be its minimum. Similarly, we assign a grid edge e of R, the minimum of the minimum elevations on the two cells adjacent to e.

Algorithm. Observe that the grid of R naturally defines a directed grid graph where each vertical cell edge is directed upwards and each horizontal cell edge is directed to the right (note that this graph is different than the VE graph). We now weight the edges of this graph according the above "flattened" elevation function. Now compute the shortest path on this weighted grid graph going from bottom left to top right. We claim the resulting quantity $\ell(\pi, \sigma)$ is a lower-bound on the sweep distance between the two curves.

Lemma 5.22. $\ell(\pi, \sigma) \leq \mathsf{d}_{SW}(\pi, \sigma)$.

Proof: Consider computing the sweep distance for this "elevation" function. Since the sweep distance is decomposed along its x and y components, and the function is a constant inside a grid cell, one can safely assume the optimal morphing is axis aligned. The only remaining possibility is that the optimal morphing enters a column on a left edge, climbs in a stairway, and leaves through an edge on the right. Let $e_1, e_2, \ldots, e_k$ be the horizontal edges this path intersects, and observe that one can always modify this path, to move vertically to the
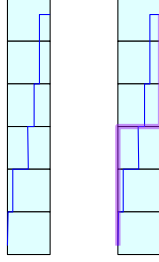
Figure 5.3: Snapping the path to the grid.

cheapest edge (under the constant elevation function), and cross using this function. Thus, the optimal path in this case can be restricted to use the grid edges, see Figure 5.3. The case that the path does the same thing in a row can be handled similarly. ■

Lemma 5.23. *For any two curves π and σ, we have $\lim_{i \to \infty} \ell(\pi^i, \sigma^i) = \mathsf{d}_{CDTW}(\pi, \sigma)$.*

Proof: The same integration argument works – the difference in value between the computed value for π^i and σ^i, and the optimal CDTW distance shrinks as i increases, as the function being integrated "converges" to a constant in the grid cells of the refined free space diagram. ■

5.5.4. Discussion of the two algorithms

Experiments show that the upper bound computed by the algorithm described above is a good approximation to the optimal solution (even with no refinement). This holds because some of the VE graph edges (specifically, left-to-right, and bottom-to-top) realize the minimum lines in the cell of the grid, thus at least one of the two integrals is minimized if the solution follow this segment, see Observation A.2. Thus, the paths suggested by the VE graph are already "cheap", at least locally.

The lower bound algorithm is far from being very good, as far as convergence after refinement. We leave the question of a better algorithm for computing a lower bound as an open problem for further research.

5.6. Fast output-sensitive simplification extractor

Given a curve π, the task at hand is to preprocess it, so that given a parameter μ, one can quickly extract a subcurve σ of π, induced by a subset of its vertices, such that the Fréchet distance between σ and π is at most μ. Furthermore, since we are dealing with huge input curves, the query time has to be proportional to $|V(\sigma)|$ – namely, the query time is output sensitive, not effected directly by the input size. We first describe an algorithm that given a curve with n vertices, in $O(n \log n)$ time, computes an array of numbers of size n, such that one can extract quickly an output-sensitive simplification from it. We then discuss ways to improve the quality of this simplification.

5.6.1. Fast extractor

A fast approximation. For a closed set $X \subseteq \mathbb{R}^d$ and a point $p \in \mathbb{R}^d$, let $d(p, X) = \min_{t \in X} \|pt\|$ be the ***distance*** of p to X. Similarly, let $\Downarrow_X(p) = \arg\min_{t \in X} \|pt\|$ be the ***nearest point*** of X to p. The ***projection*** of a set Y to a closed set $X \subseteq \mathbb{R}^d$, is the set $\Downarrow_X(Y) = \{\Downarrow_X(p) \mid p \in Y\}$.

Lemma 5.24. *Given a (directed) curve $\pi = p_1 \ldots p_n$ with n vertices, consider any (directed) segment $\tau = zz'$, such that $\Downarrow_\tau(p_1) = z$ and $\Downarrow_\tau(p_n) = z'$. One can compute in linear time a monotone morphing m, and its width $\omega(m)$, between π and τ, such that $\mathsf{d}_{\mathcal{F}}(\pi, \tau) \leq \omega(m) \leq 3\mathsf{d}_{\mathcal{F}}(\pi, \tau)$.*

Proof: Consider first the morphing m', that maps each point of π to its nearest point on τ. Clearly, $\omega(m') \leq \omega(m^\star)$, where m^* is the optimal (monotone) morphing between π and τ. Consider the monotone version m of m' that one gets from m' by never moving back along τ. If m' is monotone, then $m = m'$ and $\omega(m) = \mathsf{d}_{\mathcal{F}}(\pi, \tau)$, and we are done.
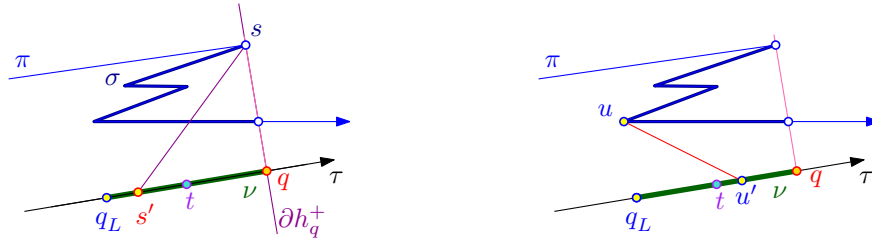

Figure 5.4: Illustration of the proof of Lemma 5.24.

The curve π is uniformly parameterized on the interval $I = [0, \|\pi\|]$, see Definition 1.1. The morphing m' matches $\pi(t)$ with the point $\Downarrow_\tau(\pi(t))$.

Let $f(t) = \|\Downarrow_\tau(\pi(t)) - z\|$ be the normalized parameterization of this projection point on τ. Thus, for all t, $f(t) \in [0, \|\tau\|]$, $f(0) = 0$, and $f(\|\pi\|) = \|\tau\|$. The *max* version of f is $F(t) = \max_{x:[0,t]} f(x)$, and the morphing m, for all t, maps $\pi(t)$ to $\tau(F(t))$. A maximal interval $J = [\alpha, \beta] \subseteq [0, \|\pi\|]$, with $\alpha < \beta$, such that $F(J)$ is a single value, is a *plateau*. For a plateau the morphing m matches all the points of $\pi(J)$ to the point $q = \tau(F(J))$. A point $x \in [0, \|\pi\|]$ not contained in any plateau, is matched by m, to the same point it was matched by m', that is $\tau(f(x)) = \Downarrow_\tau(\pi(x))$.

The only places where the leash used by m might be larger than $m^\star$ is in the plateaus. So consider such a plateau $J = [\alpha, \beta]$, with $\alpha < \beta$. Let $\nu = \{\tau(f(t)) \mid t \in J\}$ be the portion of τ that is being used by m' for points of $\sigma = \pi(J)$. The start of this plateau in π is $s = \pi(\alpha)$, and the end is $t = \pi(\beta)$. The left endpoint of $\nu \subseteq \tau$ is the projection of some point $u = \pi(\gamma)$, with $\gamma \in (\alpha, \beta)$, such that $f(J) = [f(\gamma), f(\beta)]$. Let $q_L = \tau(f(\gamma))$ and $q = \tau(f(\beta))$ be the right and left endpoints of ν, respectively, see Figure 5.4. Finally, let r be the middle point of ν.

If the optimal (monotone) morphing $m^\star$ matches s to a point $s' \in \tau$ that appears before r on τ, then $\omega(m^\star) \geq \|ss'\| \geq \|\nu\|/2$, see Figure 5.4 (left). Otherwise, $m^\star$ maps u to a

point $u' \in \tau$ that appears after r on τ. But then, $\omega(m^\star) \geq \|uu'\| \geq \|\nu\| / 2$, see Figure 5.4 (right). We conclude that

$$\omega(m^\star) \geq \max(\omega(m'), \|\nu\| / 2).$$

As a reminder, m matches all the points of $\sigma = \pi(J)$ to q. Let $\ell = \max_{p \in \sigma} d(p, \nu)$ be the maximum leash length deployed by m' for any point of σ. We have that the width of m restricted to σ is

$$\ell_J = \max_{p \in \sigma} \|pq\| \leq \sqrt{\ell^2 + \|\nu\|^2} \leq \sqrt{\omega(m')^2 + \|\nu\|^2} \leq \sqrt{2} \max(\omega(m'), \|\nu\|)$$
$$\leq 2\sqrt{2} \max(\omega(m'), \|\nu\| / 2) \leq 3\omega(m^\star).$$

Now, $\omega(m)$ is the maximum of $\omega(m')$ and the maximum of ℓ_J, over all plateaus J, which implies the claim. ∎

Preprocessing. The idea is to build a hierarchical representation of the curve. So let the input curve be $\pi = p_1 p_2 \ldots p_n$ – the output will be an array $A[1 \ldots n]$ of real numbers. The roughest approximation for π is the *spine* $p_1 p_n$. If we want a finer approximation, the natural vertex to add is $p_{\alpha(1,n)}$, where $k = \alpha(i,j) = \lfloor (i+j)/2 \rfloor$, which yields the curve $p_1 p_k p_n$. The recursive algorithm **compProfile**$(1, n)$ uses the fast approximation algorithm of Lemma 5.24 to compute

$$A[k] = \max(D(\pi[1, k]), D(\pi[k, n])).$$

The algorithm now continues filling the array recursively, by calling **compProfile**$(1, k)$ and **compProfile**(k, n).

Extracting the simplification (extract). Given a parameter w, the algorithm performs a recursive traversal of the curve, as described above. If the traversal algorithm arrives to an interval $[\![i : j]\!]$, with $k = \alpha(i, j)$, such that $A[k] > w$, then the algorithm adds p_k to the simplified curve, after extracting the simplification recursively on the range $[\![i : k]\!]$, and before extracting the simplification recursively on $[\![k : j]\!]$. If $A[k] \leq w$, then the algorithm just adds p_k to the simplification, without performing the recursive calls.

Lemma 5.25. *Given a polygonal curve π with n vertices, the algorithm **compProfile** preprocess it, in $O(n \log n)$ time, such that given a parameter $w \geq 0$, the query algorithm **extract** computes a subcurve $\overline{\pi}$ (induced by a subset of the vertices of π), such that $\mathsf{d}_{\mathcal{F}}(\pi, \overline{\pi}) \leq w$. The extraction algorithm works in $O(|V(\overline{\pi})|)$ time.*

Proof: The running time bounds are immediate. As for the correctness, it follows by observing that the simplification being output breaks the input curve into sections, such that each section is matched to a segment, such that the Fréchet distance between a section and its corresponding segment is at most w. ∎

Remark 5.26. Note, that the above just extracts a simplification quickly – this simplification might potentially have many more vertices than necessary, but in practice it works quite well. We can apply the algorithm described next as a post-processing stage to reduce the number of vertices.

35

5.6.2. A greedy simplification.

A better simplification algorithm, that yields fewer vertices than the δ-simplification used above, and the fast extractor used above, is to do a greedy approximation as suggested by Aronov *et al.* [AHK+06]. Let $\pi = p_1 \ldots p_n$. The algorithm initially starts at the vertex p_1. Assume that it is currently at the vertex p_j. The algorithm then sets the next vertex in the simplification to be p_k, where k is the smallest index greater than j such that $\mathsf{d}_{\mathcal{F}}(\pi[p_j, p_k], p_j p_k) \leq \delta$ and $\mathsf{d}_{\mathcal{F}}(\pi[p_j, p_{k+1}], p_j p_{k+1}) > \delta$ (or $k = n$ if the end of the curve is reached). One can use the algorithm Lemma 5.24 coupled with exponential and binary search to compute k in time $O((k - j + 1) \log(k - j + 1))$ (of course, here the guarantee is somewhat weaker). This algorithm yields quite a good approximation in practice with fewer vertices than the δ-simplification.

Lemma 5.27. *Given a curve π with n vertices, and a parameter δ, the above algorithm computes, in $O(n \log n)$ time, a curve $\overline{\pi}$, such that $\mathsf{d}_{\mathcal{F}}(\pi, \overline{\pi}) \leq \delta$.*

To get a better simplified curve quickly, after preprocessing, given a parameter $\delta \geq 0$, one can first use the fast extractor (**extract**) to compute a simplification (with distance $\leq \delta/10$), and then apply the above algorithm to the resulting curve to compute a simplification with distance $\leq 0.9\delta$. By the triangle inequality, the resulting simplification is distance at most δ from the original curve. This combined simplification approach works quite well in practice.

Remark 5.28. Somewhat confusingly, we have three simplification algorithms described in this paper. The first, Algorithm 5.8, is a simple linear scan of the input curve, and is potentially slow if the input curve is huge, particularly if we need to generate many simplifications of the same curve. Furthermore, in practice, the upper bound it provides on the error is way bigger than the true error. The second approach, using **compProfile** for preprocessing, and using **extract**, is much faster but still yields (in practice) inferior simplifications. Using the greedy simplification algorithm enables us to "cleanup" the simplification, and get a curve with significantly fewer vertices.

6. Conclusions and future work

In this work, we have demonstrated that our variant of the Fréchet distance is both theoretically efficient and viable in practice. We provided standard libraries for the algorithms in `Julia` and `Python`. Furthermore, our libraries compute the retractable Fréchet distance/morphing, which seems to perform quite well in practice, and is better at handling noise than the original Fréchet distance. This suggests that our algorithm may be useful in applications where the runtime is dominated by Fréchet distance computations. More generally, we believe the high quality of our implementations, and their easy availability via standard libraries in `Julia` and `Python` should make our code and algorithms easily accessible for casual users.

Clustering. Using the implementation given by Bringmann *et al.* [BKN21], there has been substantial empirical work on efficient clustering of curves. Notably, Buchin *et al.* [BDLN19] empirically study algorithms for (k, ℓ)-center clustering. We omit the formal definition of this problem here, but of note is the large number of Fréchet distance computations required to determine the furthest curve from the set of existing curves chosen as centers. We believe that our algorithm is very well suited for this task, as our heap-based implementation can be easily modified to stop early if the Fréchet distance goes beyond a given threshold.

Similarly, there has been prior empirical work on (k, ℓ)-median clustering [BBK+20] under the dynamic time warping distance, and we believe our algorithm would perform well in this application as well.

Polygonal hierarchies and fast simplification. A strategy that works quite well for the task of Section 2.2.2 is to precompute, for all input curves, the fast simplification extractor of Lemma 5.25. Then, given a query, one can simplify the two curves quickly to the right resolution δ (initially guessed to be some large value) using this extractor, then apply the algorithm of Lemma 5.27 to get a simplification with few vertices. With these simplifications, then compute the Fréchet distance, and repeat the process, with smaller δ, till the query is resolved. By caching the simplifications of the input curves, one can achieve better speedup if there are multiple computations involving the same curve.

References

[AG95]   H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *Int. J. Comput. Geom. Appl.*, 5: 75–91, 1995.

[AHK+06]   B. Aronov, S. Har-Peled, C. Knauer, Y. Wang, and C. Wenk. Fréchet distance for curves, revisited. *Proc. 14th Annu. Euro. Sympos. Alg.* (ESA), vol. 4168. 52–63, 2006.

[BBD+17]   K. Buchin, M. Buchin, D. Duran, B. T. Fasy, R. Jacobs, V. Sacristán, R. I. Silveira, F. Staals, and C. Wenk. Clustering trajectories for map construction. *Proc. 25th ACM SIGSPATIAL Int. Conf. Adv. Geog. Info. Sys., GIS,* 14:1–14:10, 2017.

[BBK+20]   M. Brankovic, K. Buchin, K. Klaren, A. Nusser, A. Popov, and S. Wong. (k, l)-medians clustering of trajectories using continuous dynamic time warping. *Proceedings of the 28th International Conference on Advances in Geographic Information Systems,* 99–110, 2020.

[BBL+16]   K. Buchin, M. Buchin, R. van Leusden, W. Meulemans, and W. Mulzer. Computing the fréchet distance with a retractable leash. *Discret. Comput. Geom.*, 56(2): 315–336, 2016.

[BBMM17]   K. Buchin, M. Buchin, W. Meulemans, and W. Mulzer. Four soviets walk the dog: improved bounds for computing the Fréchet distance. *Discrete Comput. Geom.*, 58(1): 180–216, 2017.

[BBMS19]   K. Buchin, M. Buchin, W. Meulemans, and B. Speckmann. Locally correct fréchet matchings. *Comput. Geom. Theory Appl.*, 76: 1–18, 2019.

[BDLN19]   K. Buchin, A. Driemel, N. van de L'Isle, and A. Nusser. Klcluster: center-based clustering of trajectories. *Proc. 27th ACM SIGSPATIAL Int. Conf. Adv. GIS,* 496–499, 2019.

[BKN21]   K. Bringmann, M. Künnemann, and A. Nusser. Walking the dog fast in practice: algorithm engineering of the fréchet distance. *J. Comput. Geom.*, 12(1): 70–108, 2021.

[CKT+16]   S. Chechik, H. Kaplan, M. Thorup, O. Zamir, and U. Zwick. Bottleneck paths and trees and deterministic graphical games. *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France,* vol. 47. 27:1–27:13, 2016.

[DHW12]   A. Driemel, S. Har-Peled, and C. Wenk. Approximating the Fréchet distance for realistic curves in near linear time. *Discrete Comput. Geom.*, 48(1): 94–127, 2012.

[GT88]   H. N. Gabow and R. E. Tarjan. Algorithms for two bottleneck optimization problems. *J. Algorithms*, 9(3): 411–417, 1988.

[HMW+20]   C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825): 357–362, 2020.

[HR14]   S. Har-Peled and B. Raichel. The Fréchet distance revisited and extended. *ACM Trans. Algo.*, 10(1): 3:1–3:22, 2014.

[LPS15]   S. K. Lam, A. Pitrou, and S. Seibert. Numba: a llvm-based python jit compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC,*

[Max23]   Maxima. *Maxima, a Computer Algebra System. Version 5.47.0.* https://maxima.sourceforge.io/, 2023.

[MP99]   M. E. Munich and P. Perona. Continuous dynamic time warping for translation-invariant curve alignment with applications to signature verification. *Proceedings of the International Conference on Computer Vision, Kerkyra, Corfu, Greece, September 20-25, 1999,* 108–115, 1999.

[PGR+17]   E. Pollonara, T. Guilford, M. Rossi, V. Bingman, and A. Gagliardo. *Data from: Right hemisphere advantage in the development of route fidelity in homing pigeons*. 2017.

[RKT+18a]   S. Rotics, M. Kaatz, S. Turjeman, D. Zurell, M. Wikelski, N. Sapir, U. Eggers, W. Fiedler, F. Jeltsch, and R. Nathan. *Data from: Early arrival at breeding grounds: causes, costs and a trade-off with overwintering latitude*. 2018.

[RKT+18b]   S. Rotics, M. Kaatz, S. Turjeman, D. Zurell, M. Wikelski, N. Sapir, U. Eggers, W. Fiedler, F. Jeltsch, and R. Nathan. Early arrival at breeding grounds: causes, costs and a trade-off with overwintering latitude. *J. Animal Ecology*, 87(6): 1627–1638, 2018.

[Wil08]   B. Williams. *Character Trajectories*. UCI Machine Learning Repository. 2008.

[WO18]   M. Werner and D. Oliver. ACM SIGSPATIAL GIS cup 2017: range queries under fréchet distance. *ACM SIGSPATIAL Special*, 10(1): 24–27, 2018.

[ZFX+11]   Y. Zheng, H. Fu, X. Xie, W.-Y. Ma, and Q. Li. *Geolife GPS trajectory dataset - User Guide*. Geolife GPS trajectories 1.1. July 2011.

A. The elevation function

We need some standard properties of the elevation function. We prove some of them here, so that our presentation would be self contained.

A.1. A helper lemma

Lemma A.1. *for $f : \mathbb{R}^k \to \mathbb{R}^d$ an affine function, the function $u(q) = \|f(q)\|$ is convex.*

Proof: The proof is straightforward, and the reader is encouraged to skip reading it. Fix any two points $\mathsf{p}'.\mathsf{q}' \in \mathbb{R}^k$, and consider the segment $\mathsf{p}'\mathsf{q}'$. We need to prove that $h(t) = \left\| f((1-t)\mathsf{p}' + t\mathsf{q}') \right\|$ is convex. Let $\mathsf{p} = f(\mathsf{p}')$ and $\mathsf{q} = f(\mathsf{q}')$. Since f is affine, we have that

$$h(t) = \left\| f((1-t)\mathsf{p}' + t\mathsf{q}') \right\| = \left\| (1-t)f(\mathsf{p}') + tf(\mathsf{q}') \right\| = \| (1-t)\mathsf{p} + t\mathsf{q} \| = \sqrt{\sum_{i=1}^{d} g_i(t)},$$

where $g_i(t) = ((1-t)p_i + tq_i)^2 = \alpha_i t^2 + \beta_i t + \gamma_i$, $\alpha_i, \beta_i, \gamma_i$ are constants, for $i = 1, \dots, d$, $\mathsf{p} = (p_1, \dots, p_d)$, and $\mathsf{q} = (q_1, \dots, q_d)$. For any i, the function $g_i(t)$ is nonnegative. If $\alpha_i = 0$ then $g_i(t) = \gamma_i$ and then $p_i = q_i$. Otherwise, $\alpha_i > 0$ and $g_i(t)$ is a parabola. Let $\alpha = \sum_i \alpha_i$, $\beta = \sum_i \beta_i$ and $\gamma = \sum_i \gamma_i$. Consider the function $g(t) = \sum_{i=1}^{d} g_i(t) = \alpha t^2 + \beta t + \gamma$. If, for all i, $p_i = q_i$, then $g(t) = \gamma$, and the claim trivially holds. Otherwise, $g(t)$ is a non-negative parabola with $\alpha > 0$, and since it has at most a single root, we have that $\beta^2 - 4\alpha\beta \leq 0$.

Now, we have

$$h'(t) = \frac{2\alpha t + \beta}{2\sqrt{\alpha t^2 + \beta t + \gamma}} = \frac{h_1(t)}{h(t)} \qquad \text{for } h_1(t) = \alpha t + \beta/2,$$

$$\text{and} \qquad h''(t) = \frac{h(t)g'(t) - h'(t)h_1(t)}{(h(t))^2} = \frac{\alpha h(t) - (h_1(t))^2/h(t)}{(h(t))^2} = \frac{(h(t))^2 - (h_1(t))^2/\alpha}{(h(t))^3/\alpha}.$$

This implies that

$$\text{sign}(h''(t)) = \text{sign}\left((h(t))^2 - (h_1(t))^2/\alpha\right) = \text{sign}\left(\alpha t^2 + \beta t + \gamma - \alpha t^2 - \beta t - \beta^2/4\alpha\right)$$
$$= \text{sign}\left(\gamma - \beta^2/4\alpha\right) = \text{sign}\left(4\alpha\gamma - \beta^2\right) \geq 0,$$

since $\alpha > 0$ and $\beta^2 - 4\alpha\gamma \leq 0$. Which implies that $h(\cdot)$ is convex, and so is $\|f(\cdot)\|$. ∎

A.2. Back to the elevation function

We are given two segments $p_0 p_1$ and $q_0 q_1$, where $L_p = \|p_0 p_1\|$ and $L_q = \|q_0 q_1\|$. Their uniform parameterization is

$$f(s) = p_0 + s\bar{p} \qquad \text{and} \qquad g(t) = q_0 + t\bar{q},$$

where $\bar{p} = (p_1 - p_0)/L_p$ and $\bar{p} = (p_1 - p_0)/L_q$. The elevation function is

$$\forall (s,t) \in [0, L_p] \times [0, L_q] \qquad e(s,t) = \|u_0 + s\bar{p} - t\bar{q}\|,$$

where $u_0 = p_0 - q_0$. Since the elevation function is the norm of an affine function, it is convex, by Lemma A.1.

The squared elevation function is

$$E(s,t) = (e(s,t))^2 = \langle u_0 + s\bar{p} - t\bar{q}, u_0 + s\bar{p} - t\bar{q}\rangle$$
$$= \|u_0\|^2 + 2\langle u_0, s\bar{p} - t\bar{q}\rangle + s^2 - 2st\langle \bar{p}, \bar{q}\rangle + t^2,$$
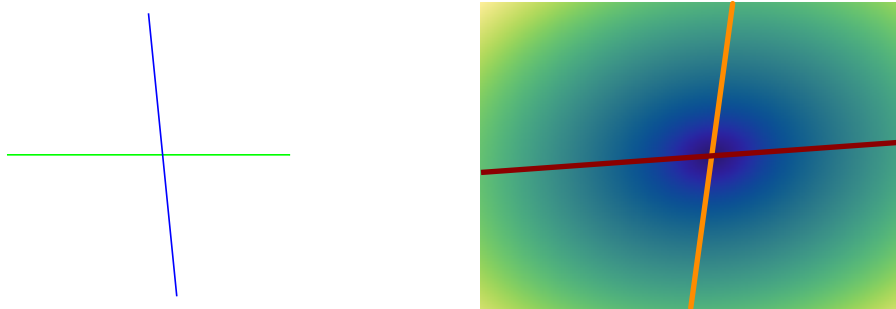

Figure A.1: The elevation function for two segments, and the minimum lines. See relevant animations here.

since $\|p\|^2 = 1$ and $\|q\|^2 = 1$. Its **_derivative_** along s and t respectively is

$$\partial_s E(s,t) = 2 \langle u_0, p \rangle + 2s - 2t \langle p, q \rangle$$
$$\text{and} \quad \partial_t E(s,t) = -2 \langle u_0, q \rangle - 2s \langle p, q \rangle + 2t$$

In particular, let $h(\alpha)$ (resp. $v(\beta)$) be the s (resp. t) coordinate of the minimum of the elevation function on the horizontal line $t = \alpha$ (resp. vertical line $s = \beta$). We have that $h(\alpha)$ is the solution to the (linear) equation $\partial_s E(s, \alpha) = 0$ (resp. $\partial_t E(\beta, t) = 0$). That is

$$h(\alpha) = \langle p, q \rangle \alpha - \langle u_0, p \rangle \quad \text{and} \quad v(\beta) = \beta \langle p, q \rangle + \langle u_0, q \rangle.$$

Observation A.2. *In particular, the edges connecting the left portal to the right portal, and the edge connecting the bottom portal of a cell to its top portal, are both tracing these minimum edges.*