Efficient trajectory queries under the Fréchet distance (GIS Cup)

Efficient trajectory queries under the Fréchet distance (GIS Cup)

Kevin Buchin
TU Eindhoven, the Netherlands
k.a.buchin@tue.nl

Yago Diez
Tohoku University, Japan
yago@dais.is.tohoku.ac.jp

Tom van Diggelen
TU Eindhoven, the Netherlands
t.w.t.v.diggelen@student.tue.nl

Wouter Meulemans
TU Eindhoven, the Netherlands
w.meulemans@tue.nl

ABSTRACT

Consider a set $\mathcal{P}$ of trajectories (polygonal lines in $\mathbb{R}^2$), and a query given by a trajectory Q and a threshold $\varepsilon > 0$. To answer the query, we wish to find all trajectories $P \in \mathcal{P}$ such that $\delta_F(P, Q) \leq \varepsilon$, where δ_F denotes the Fréchet distance. We present an approach to efficiently answer a large number of queries for the same set $\mathcal{P}$. Key ingredients are (a) precomputing a spatial hash that allows us to quickly find trajectories that have endpoints near Q; (b) precomputing simplifications on all trajectories in $\mathcal{P}$; (c) using the simplifications and optimizations of the decision algorithm to efficiently decide $\delta_F(P, Q) \leq \varepsilon$ for most $P \in \mathcal{P}$.

CCS CONCEPTS

• **Information systems → Geographic information systems**;
• **Theory of computation → Computational geometry**;

KEYWORDS

Fréchet distance, trajectories, range searching

ACM Reference Format:
Kevin Buchin, Yago Diez, Tom van Diggelen, and Wouter Meulemans. 2017. Efficient trajectory queries under the Fréchet distance (GIS Cup). In *Proceedings of SIGSPATIAL'17*. ACM, New York, NY, USA, 4 pages.
https://doi.org/10.1145/3139958.3140064

1 INTRODUCTION

With GPS sensors becoming low-cost and near-ubiquitous, trajectory data has seen a rapid increase in volume. A central problem, underlying many trajectory analyses in one way or another, is the need to quantify trajectory similarity and to find trajectories similar to a given one. A common way to quantify trajectory similarity is the *Fréchet distance* [3], which has been used to, e.g., find similar subtrajectories [4] or detect interaction between moving objects [5].

To define this metric, consider two trajectories (polygonal lines), $P = \langle p_0, \ldots, p_m \rangle$ and $Q = \langle q_0, \ldots q_n \rangle$, defined by their vertices. We interpret these as functions $P\colon [0, m] \to \mathbb{R}^2$ using $P(i + \lambda) = (1 - \lambda)p_i + \lambda p_{i+1}$, and likewise for Q. Now, consider the set Ψ of all orientation-preserving homeomorphisms between the two parameter spaces, that is, of continuous functions $\psi\colon [0, m] \to [0, n]$ with a continuous inverse such that $\psi(0) = 0$ and $\psi(m) = n$. The Fréchet distance is now defined as follows:

$$\delta_F(P, Q) = \inf_{\psi \in \Psi} \max_{t \in [0,m]} \|P(t) - Q(\psi(t))\|$$

The challenge. The GIS Cup 2017 sets out the following challenge. Consider a database $\mathcal{P}$ of trajectories, together with a set $\mathcal{Q}$ of range searching queries [1], where each query is a pair (Q, ε) of a trajectory Q and a threshold $\varepsilon > 0$. For each query (Q, ε), we are tasked to find all trajectories $P \in \mathcal{P}$ with $\delta_F(P, Q) \leq \varepsilon$. In particular, $\mathcal{P}$ and $\mathcal{Q}$ are sufficiently large for preprocessing to pay off.

Contributions. We provide an efficient approach to solve the given challenge, by combining and optimizing a number of techniques. The key ingredients are spatial hashing, simplification, and optimized exact decision procedures. We first provide a high-level overview of our algorithm, followed by a detailed explanation of each component. The implementation is available online[1].

2 ALGORITHM OVERVIEW

The algorithm solves the GIS Cup problem in two stages: a preprocessing stage, followed by a query stage. The preprocessing stage consists of building two main data structures: a spatial hash and four levels of simplifications for each trajectory in $\mathcal{P}$. The query stage uses these two data structures determine for each $P \in \mathcal{P}$ whether $\delta_F(P, Q) \leq \varepsilon$, for each query (Q, ε). Each query is processed in four steps, as summarized below.

(1) We first use the spatial hash to obtain a candidate set $C \subseteq \mathcal{P}$, which contains all trajectories whose start and endpoints are within distance ε of the start and endpoints of Q.
(2) For each trajectory $P \in C$, we now do the following steps, stopping as soon as we know how $\delta_F(P, Q)$ compares to ε.
   (a) Starting with the most coarse level, we compare the simplifications of P and Q to try to decide whether $\delta_F(P, Q) \leq \varepsilon$.
   (b) An imprecise but fast greedy algorithm is run to check if $\delta_F(P, Q) \leq \varepsilon$ for cases in which a simple linear homeomorphism suffices (equal-time distance).
   (c) We run our exact continuous decision Fréchet algorithm to establish whether $\delta_F(P, Q) \leq \varepsilon$. This is always conclusive.

The key idea of the above is to spend as little time as possible to establish whether $P \in \mathcal{P}$ is within Fréchet distance ε of a trajectory Q. Each consecutive step takes more time, but is more precise. The

[1] https://github.com/TWTDIG/GISCUP17TUE

SIGSPATIAL'17, November 7–10, 2017, Los Angeles Area, CA, USA

ACM ISBN 978-1-4503-5490-5/17/11…$15.00
https://doi.org/10.1145/3139958.3140064

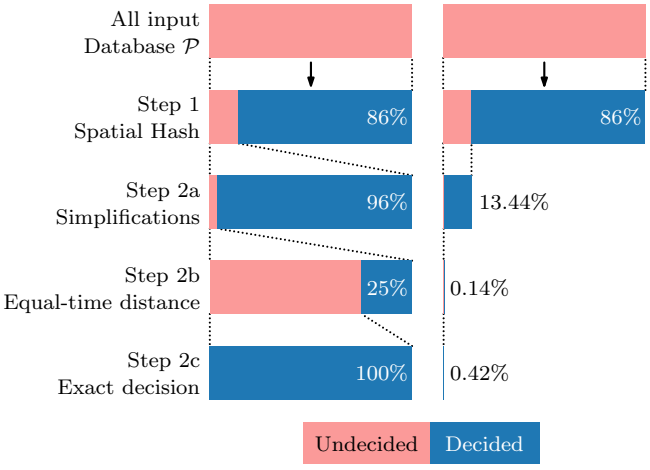

Figure 1: The pruning percentages of each step, as a fraction of the step's input (left) and of the total database (right).

earlier steps are powerful enough to filter out a majority of the trajectories. In particular, only very few trajectories require Step 2c, which could in theory take quadratic time, whereas the other steps are linear or use significantly simplified trajectories. We use parallelism to speed up computation where possible, using a number of parallel tasks up to what the system can handle.

Figure 1 shows the pruning percentages on our test data set[2]. We constructed this data set, using a database $\mathcal{P}$ consists 3200 trajectories randomly sampled from the challenge sample trajectory set. The corresponding query set $\mathcal{Q}$ consists of 6400 trajectories randomly selected from the challenge sample trajectory set, with ranges ε generated such that the average query returns 2% of $\mathcal{P}$, while rare queries return no trajectories or up to 30% of $\mathcal{P}$. This makes the queries fairly realistic while also containing exceptional cases. In our tests, the algorithm solves the problem in 7 seconds, which is 30 times faster compared to using no pruning steps (i.e. only using Step 2c). Total processing time increases from 7 to 94 seconds when using fewer to no simplification levels, while reducing the preprocessing time by only 200 milliseconds per omitted level. Despite the observed effectiveness of simplification, we chose not to add a fifth level: it effects the memory footprint, and performance gain decreases for additional levels.

In the next sections, we provide details for each step. Throughout, we assume that a database trajectory $P \in \mathcal{P}$ has $m+1$ vertices, $p_0, \ldots, p_m$; a query trajectory Q has $n+1$ vertices, $q_0, \ldots, q_n$.

3 A SPATIAL HASH

During the preprocessing stage, we create a spatial hash. It is implemented as a regular grid, dividing the bounding box (minimal axis-aligned rectangle enclosing all trajectories in $\mathcal{P}$) in 500 equally-sized rows and columns, yielding 250000 cells in total. Each cell c contains a list of vertices that have coordinates within c. We place p_0 and p_m into the spatial hash for each trajectory $P \in \mathcal{P}$. The insertions take only $O(1)$ time per trajectory and end up as a relatively insignificant amount of time with respect to the total execution time. Though parallelization could be considered, the need for synchronization in accessing the list of a cell makes for only a very small potential gain and we hence did not use concurrency here.

During the query stage, we need the spatial hash only to establish the set C of candidates with close-enough endpoints to Q in Step 1. We retrieve the trajectories $P \in \mathcal{P}$ with $\|p_0 - q_0\| \leq \varepsilon$ or with $\|p_m - q_n\| \leq \varepsilon$, by using the vertices stored with cells in the hash that are close enough to q_0 and q_n respectively. We then intersect these lists to obtain C. Note that any trajectory with $\|p_0 - q_0\| > \varepsilon$ or $\|p_m - q_n\| > \varepsilon$ cannot have $\delta_F(P, Q) \leq \varepsilon$, since the endpoints must map to each other in any homeomorphism; thus any trajectory not in C is indeed not part of the answer for query (Q, ε).

4 SIMPLIFICATIONS

In the preprocessing stage, we simplify each trajectory $P \in \mathcal{P}$. For this we use the algorithm by Agarwal et al. [2] which computes a simplification T' of some trajectory T with $\delta_F(T, T') \leq \delta$, given some threshold $\delta > 0$. There is a direct trade off between δ and the number of vertices of T'; that is, between the power and computation time of subsequent comparisons in the query stage. We therefore construct four simplifications, with increasing level of detail. These simplifications are thus increasingly time-consuming to compute with, but also yield more more accurate results. This way, easily decidable trajectory comparisons can be discarded with little computation, while trajectories more difficult to decide still benefit from a detailed comparison.

In particular, for each trajectory $P \in \mathcal{P}$ of size m, we run four binary searches to find threshold that achieve simplifications with $0.07m$, $0.19m$, $0.24m$, and $0.32m$ vertices. These values were determined experimentally by observing the algorithm behavior on our constructed test data set (see also Section 2). We denote the obtained simplifications by $P^0, \ldots, P^3$, where P^0 is the coarsest simplification, and P^3 the finest. These simplifications are computed in parallel per trajectory.

We can use the simplifications in the query stage, by exploiting that the Fréchet distance is a proper metric and thus the triangle inequality can be used. First, we similarly compute simplifications $Q^0, \ldots, Q^3$ for query trajectory Q. Instead of searching for exact thresholds, we estimate good thresholds using the thresholds used to simplify $\mathcal{P}$. The estimation is computed by averaging, for each trajectory, the relation between the computed threshold and the diagonal of the trajectory bounding box. This reduces computation time per query, and implicitly assuming that the trajectories from $\mathcal{P}$ and $\mathcal{Q}$ have similar characteristics and thus can be similarly simplified. For each $i = 0, \ldots, 3$, we do the following:

(I) Compute the equal-time distance d between P_i and Q_i (Section 5). If $d < \varepsilon - \delta_i - \gamma_i$, then by triangle inequality we may conclude that $\delta_F(P, Q) \leq \varepsilon$.
(II) We now use our exact decision algorithm (Section 6). If $\delta_F(P_i, Q_i) \leq \varepsilon - \delta_i - \gamma_i$, we conclude that $\delta_F(P, Q) \leq \varepsilon$.
(III) Finally, we use the exact decision algorithm a second time. If $\delta_F(P_i, Q_i) > \varepsilon + \delta_i + \gamma_i$, we can conclude $\delta_F(P, Q) > \varepsilon$.

If we reach a conclusion in any of the above steps, we stop processing P and report it for the query's answer if we conclude $\delta_F(P, Q) \leq \varepsilon$ (I or II). If we reach no conclusion, we proceed with the next more detailed level of simplification.

[2] Available online at https://github.com/TWTDIG/GISCUP17TUE

5 USING THE EQUAL-TIME DISTANCE

In Step 2a and 2b of the algorithm, we use the equal-time distance to determine an upperbound on the Fréchet distance. That is, we use a linear homeomorphism $\psi\colon [0,m] \to [0,n]$ with $\psi(t) = t \cdot n/m$, to map points on P to Q. It is straightforward to compute $d = \max_{t\in[0,m]} \|P(t) - Q(\psi(t))\|$ in linear time; and since $\psi \in \Psi$, this readily implies that $\delta_F(P,Q) \le d$. Thus, if $d \le \varepsilon$, we can immediately conclude that $\delta_F(P,Q) \le \varepsilon$ by transitivity.

6 THE FRÉCHET DECISION PROBLEM

We need an algorithm to decide whether $\delta_F(P,Q) \le \varepsilon$ in Step 2c and similarly for step 2a. The decision algorithm by Alt and Godau [3] runs in $\Theta(mn)$ time. This would be problematic for efficient processing of complex trajectories, thus especially for Step 2c—comparing the full trajectories. Below, we present an improved variant of this algorithm, used in both Step 2a and Step 2c.

The Alt and Godau algorithm conceptually works on a parameter-space representation of the problem. This is called the *free-space diagram*, a $[0,m]\times[0,n]$ diagram where a point (s,t) represents the pair of points $P(s)$ and $Q(t)$. We call a point (s,t) of the diagram *free* if $\|P(s) - Q(t)\| \le \varepsilon$. The *free space* is then the union of all free points. Alt and Godau showed that a bimonotone path from $(0,0)$ (bottomleft of the diagram) to (m,n) (topright) through the free space corresponds to a homeomorphism which acts as a witness for $\delta_F(P,Q) \le \varepsilon$ [3]. Points (s,t) that are free and reachable with such a bimonotone path are called *reachable*; the union of reachable points is called the *reachable space*. The decision problem is reduced to deciding whether (m,n) is reachable.

The vertices of P and Q partition the diagram into $m \cdot n$ cells. Alt and Godau provide a dynamic program that processes each cell to find out which points along the cell boundaries are reachable. This algorithm uses the interval of reachable points on the left and bottom side of the cell, to compute those for the top and right side.

We define the complexity of the reachable space as the number of reachable vertex pairs p_i, q_j plus the number of reachable points (s,t) for which $\|P(s) - Q(t)\| = \varepsilon$ and either $P(s)$ or $Q(t)$ is a vertex. To improve upon the quadratic run time, we design our method such that the run time is $O(mn)$, that is, proportional to the complexity of the reachable space rather than the $\Theta(mn)$ number of cells.

We first present a basic version of our algorithm, which uses queues to efficiently find the reachable space. We then describe how we re-use information from the simplification steps, to further speed up our algorithm.

6.1 Queue-based approach

Algorithm 1 provides an overview of our basic algorithm and is further explained below. For this, we assume that $m \le n$, and number the rows 1 though m (corresponding to edges of P) and the columns 1 through n (the edges of Q). Note that if $m > n$, we simply swap the roles of P and Q. We denote the reachable intervals on the left, right, top and bottom side of a cell i,j by L_{ij}, R_{ij}, T_{ij} and B_{ij} respectively. Note that $R_{ij} = L_{i(j+1)}$ and $T_{ij} = B_{(i+1)j}$.

The main idea is to extend the column-by-column dynamic program, to maintain a list of row indices with reachable intervals that we could reach in the previous column to jump over unreachable parts. That is, we use a queue Λ_j for each vertex q_j of Q, that stores

Algorithm 1 DECIDE(P,Q,ε)

Given: P and Q are trajectories with m and n edges, and $\varepsilon > 0$
Decide: whether $\delta_F(P,Q) \le \varepsilon$

```
1:  Set Λ_0 to ⟨1⟩
2:  Set j to 1
3:  while Λ_{j-1} is not empty and j ≤ n do
4:    while Λ_{j-1} is not empty do
5:      Set i to index at the head of Λ_{j-1}
6:      Set B to an empty interval
7:      repeat
8:        if the head of Λ_{j-1} stores index i then
9:          Set L to the reachable interval at the head of Λ_{j-1}
10:         Remove the head of Λ_{j-1}
11:       else
12:         Set L to an empty interval
13:       Use L and B to compute reachable intervals R and T
14:       If R ≠ ∅, add (i, R) to queue Λ_j
15:       Set B to T and increase i by one
16:     until B = ∅
17:   Increase j by one
18: return true iff m is contained in Λ_n
```

the indices of edges $p_{i-1}p_i$ of P that have a nonempty reachable interval with q_j, in sorted order. We set L_0 to $\langle 1\rangle$, as we know that $(0,0)$ is reachable. As we process a column $j = 1,\dots,n$, we use queue Λ_{j-1} and build queue Λ_j (see also Figure 2(a)).

We process a column as follows (Line 4 to 16). We immediately jump to the first index in Λ_{j-1}, knowing that any earlier cells are not reachable. We then essentially start Alt and Godau's procedure from there, going up in the column, each time processing cell i,j by computing R_{ij} and T_{ij} using L_{ij} and B_{ij} (Line 7 to 16). When we find that i matches the index stored in at the head of the queue Λ_{j-1}, we remove it and obtain the corresponding nonempty L_{ij}; otherwise, we know that L_{ij} is empty. If $R_{ij} \ne \emptyset$, we add the row index i with corresponding R_{ij} to Λ_j. We stop this process after we find that $T_{ij} = \emptyset$, that is, after we conclude that we cannot reach the next cell from below. Then, if there are still elements in Λ_{j-1}, we jump to the next index there; otherwise, we stop processing the column and move to the next, as no cell in the remainder of the current column can be reachable.

To implement the above algorithm, we observe that we only ever need Λ_{j-1} and Λ_j. We therefore implement this using only two queues, using one as Λ_{j-1} and the other as Λ_j; the queues swap roles when moving to the next column, to avoid copying queues.

We further improve this method by compressing the queues. Instead of remembering each reachable row index, we save space in the queue by instead storing maximal vertical intervals of reachable space, characterized by a start and end point (which may be part of different rows). At this point this optimization saves only space and queue operations; we still need to process all rows in between to check whether a long reachable interval splits into multiple. However, it in fact also allows us to do one more optimization, where we leverage information from our previous simplifications; this is described in the upcoming section.

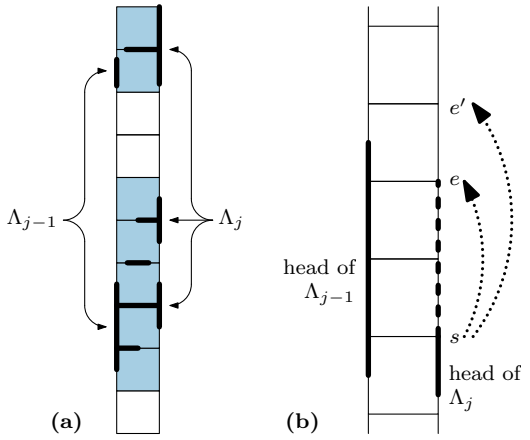

Figure 2: (a) Processing column j computes the reachable intervals on cell boundaries, shown as thick lines. Queues Λ_{j-1} and Λ_j contain the vertical intervals; those of Λ_{j-1} are used to compute those in Λ_j. Only the shaded grid cells are inspected by the algorithm. (b) Schematic illustration of two potential free-space jumps, (s, e) and (s, e'), shown as arrows. If condition 2 is met, (s, e) may be used to immediately conclude that the corresponding interval (dashed) is fully reachable. Jump (s, e') violates condition 1, as it extends beyond the reachable interval in Λ_{j-1}: it cannot be used.

6.2 Free-space jumps

We use information from our simplification steps to define *free-space jumps*. These jumps allow us to find sequences of connected reachable rows in Λ_j without processing each row individually, given certain preconditions. In our tests on the sample data sets, the application of free-space jumps reduced the number of cells processed, and so the number of steps in our algorithm by a factor of five, while requiring little additional storage or computation.

During trajectory simplification, we have calculated the Fréchet distance of many subtrajectories to the source trajectory in order to find a simplification matching the given threshold. We can use the (normally discarded) Fréchet distance of these subtrajectories to skip computations in the decision Fréchet problem. Given start and end indices (s, e) of a subtrajectory and an accompanying Fréchet distance d, we can skip the calculation of the rows between s and e in any decision Fréchet problem if the following conditions hold:

(1) Interval (s, e) is contained in a single interval in queue Λ_{j-1}.
(2) $max(\delta(j, s), \delta(j, e)) + d \leq \varepsilon$

The first condition ensures that skipping range (s, e) does not result in a loss of information. That is, by having $(j-1, e)$ reachable, we do not need to propagate the reachability information up through all these cells: the top of each of the intermediate cells is fully reachable by this condition. The second condition ensures that the jump is valid: the range (s, e) on column j is part of the free-space, and so can be skipped. This is illustrated in Figure 2(b). Skipping these calculations as described constitutes a free-space jump. As only jumps along a column are possible, only one of the trajectories needs to retain the subtrajectory Fréchet computation results.

6.3 Performance

Though a theoretic difference between $O(mn)$ and $\Theta(mn)$ may seem minor, it has a major impact on our running time. We explain this as follows. When the need arises to compare two trajectories in Step 2c, we can safely assume that the trajectories are very similar: otherwise, a previous step would have likely already been able to give us a definite answer for the comparison. With the two trajectories being very similar, the reachable-space complexity is likely much lower than quadratic; this makes our optimized version efficient, even when executed with two full trajectories. Note that this is not a hard guarantee: trajectories with a high reachable-space complexity may remain, even after all pruning steps.

7 FURTHER CONSIDERATIONS

Our method uses various parameters (e.g. spatial hash size, number of simplifications and their thresholds). We set these based on experiments with several example data sets. However, more experiments could be set up to further fine-tune these parameters and investigate how characteristics of the data sets may inform these parameter settings; the latter would allow us to move towards an approach that adapts its parameters to fit the actual data set.

We use equal-time distance as a simple greedy approach, but various alternatives could be considered. On the sample data set, it decided only 25% of the cases that made it to Step 2b. Other heuristic methods could be considered, if they have significantly better pruning percentage and/or average running time.

Our approach to the decision Fréchet problem requires little storage in practice compared to other column-based approaches. This results in efficient CPU processing, but also makes it a good candidate for a GPU-based implementation, where IO is often a bottleneck and memory is expensive.

Finally, the free-space jumps can be used to convert the decision Fréchet problem into a graph problem, where the edges are pairs of jumps and reachability is computed by traversing this graph. We expect that other algorithms may be developed in this new problem space, especially greedy approximation algorithms.

ACKNOWLEDGMENTS

The authors would like to thank Matias Korman and Maike Buchin for initial discussions. K.B. was supported in part by the Netherlands Organisation for Scientific Research (NWO, 612.001.207). Y.D. was supported by the IMPACT Tough Robotics Challenge Project of the Japan Science and Technology Agency. W.M. was supported by the Netherlands eScience Center (NLeSC, 027.015.G02).

REFERENCES

[1] Peyman Afshani and Anne Driemel. 2017. On the complexity of range searching among curves. ArXiv:1707.04789. (2017).
[2] Pankaj K. Agarwal, Sariel Har-Peled, Nabil H. Mustafa, and Yusu Wang. 2005. Near-linear time approximation algorithms for curve simplification. *Algorithmica* 42, 3-4 (2005), 203–219.
[3] Helmut Alt and Michael Godau. 1995. Computing the Fréchet Distance between two Polygonal Curves. *Int. J. of Comp. Geom. and Appl.* 5, 1 (1995), 75–91.
[4] Kevin Buchin, Maike Buchin, Joachim Gudmundsson, Maarten Löffler, and Jun Luo. 2011. Detecting Commuting Patterns by Clustering Subtrajectories. *Int. J. of Comp. Geom. and Appl. (19th ISAAC special issue)* 21, 3 (2011), 253–282.
[5] Maximilian Konzack, Thomas McKetterick, Tim Ophelders, Maike Buchin, Luca Giuggioli, Jed Long, Trisalyn Nelson, Michel A. Westenberg, and Kevin Buchin. 2017. Visual analytics of delays and interaction in movement data. *Int. J. of GIS* 31, 2 (2017), 320–345.