A Middle Curve Based on Discrete Fréchet Distance*

Hee-Kap Ahn† Helmut Alt‡ Maike Buchin§ Ludmila Scharf¶ Carola Wenk‖

Abstract

Given a set of polygonal curves we seek to find a "middle curve" that represents the set of curves. We ask that the middle curve consists of points of the input curves and that it minimizes the discrete Fréchet distance to the input curves. We develop algorithms for three different variants of this problem.

1 Introduction

Consider a group of animals or people traveling together, several of which are GPS-tagged. Based on their trajectories, i.e., sequences of time-stamped positions, we want to compute a representation of a middle path taken by the group. Because sampled locations are more reliable than positions interpolated in between those, we seek a middle path consisting only of sampled locations. The middle path should be as close as possible to the path of the individuals, hence we ask for it to minimize the discrete Fréchet distance to these. The Fréchet distance [2] and the discrete Fréchet distance [4] are well-known distance measures, which have been used before in trajectory analysis.

We consider three variants of this problem, which we introduce now more formally for two curves. Given two point sequences P and Q, of length n and m respectively, and $\varepsilon > 0$, we wish to determine whether there exists a *middle curve* R consisting of points from $P \cup Q$ with $\max(d_F(R,P), d_F(R,Q)) \leq \varepsilon$, where d_F denotes the discrete Fréchet distance.

In the following definitions we assume that each point in R uniquely corresponds to a point in P or Q (in particular, if P and Q share points). We say the middle curve R is *ordered*, if any two points of P occurring in R have the same order as in P, likewise with points from Q. We say the middle curve R is *restricted*, if points on R are mapped to themselves in a matching realizing the discrete Fréchet distance. That is, consider a point p in R originating from P; In a matching between R and P realizing $d_F(R,P)$, p as a point of R is mapped to itself on P.

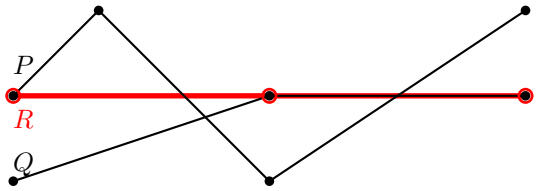

Figure 1: Example of a middle curve R of curves P, Q.

Related work. Several papers [3, 5] study the problem of finding a middle curve but without the restriction that the middle curve should consist of points of the input curves. Buchin et al. [3] restrict to use parts of edges of the input, and the aim is to always "stay in the middle" in the sense of a median. Har-Peled and Raichel [5] show that without any restrictions on the middle curve (i.e., neither using input vertices nor edges), a curve minimizing the Fréchet distance to k input curves can be computed in the k-dimensional free space using the radius of the smallest enclosing disk as "distance".

2-**Approximation.** A simple observation is that choosing any of the input curves is a 2-approximation to minimizing the distance (using the triangle inequality). Thus, we have a 2-approximation in constant time (not counting the time to output the points of the curve). Also, it is easy to give an example showing that this 2-approximation is tight.

Results. We develop algorithms for three variants of this problem (runtime for $k \geq 2$ curves of size at most n each):

1. An $O(n^k + n^2 \log n)$ time algorithm for computing an unordered middle curve,
2. An $O(n^{2k})$ time algorithm for computing an ordered middle curve,
3. An $O(n^k \log^{k-1} n)$ time algorithm for computing an ordered and restricted middle curve.

In the following, we will also call these three cases the "unordered, ordered, and restricted case". In the following sections, we present these algorithms. Due to space restrictions, we focus on describing the algorithms, omitting details and proofs.

*This work was partially supported by research grant AL 253/8-1 from Deutsche Forschungsgemeinschaft (German Science Association), and by the National Science Foundation under grant CCF-1301911.

†POSTECH, heekap@postech.ac.kr
‡Free University of Berlin, alt@mi.fu-berlin.de
§Ruhr University Bochum, maike.buchin@rub.de
¶Free University of Berlin, scharf@mi.fu-berlin.de
‖Tulane University, cwenk@tulane.edu

2 Algorithm for the unordered case

To solve the decision problem for the unordered case, we modify the algorithm for computing the discrete Fréchet distance of two curves [4] as follows. We search again for a path in the free space matrix. Now (in contrast to the original algorithm) we color a vertex (i, j) free iff there exists any vertex v from P or Q such that v has distance $\leq \varepsilon$ to both p_i and q_j. Then again we search for a monotone path in the free space matrix. For the computation problem, we label each vertex (i, j) with $\min_{v \in P \cup Q} \max(\|v - p_i\|, \|v - q_j\|)$, and search for a path minimizing the maximum label.

The runtime for searching the grid is (in both cases) $O(mn)$. To compute the vertex labels (0|1 or distances) takes $O(mn(m+n))$ time brute-force (i.e., for each vertex (i, j) test all $(m + n)$ possibilities for v in $O(1)$ time). For k curves of length at most n this takes $O(kn^{(k+1)})$ time in total. Next, we describe how to do this more efficiently. Here, we use a circular sweep to determine for each point p all points q such that (p, q) is free, i.e., there is some point v of P or Q which has distance $\leq \varepsilon$ to both p and q.

Constructing the free space matrix. For any $p \in P$:

1. Determine all disks of radius ε around points in $P \cup Q$ that contain p.
2. Determine the union U of those disks. This can be done by divide-and-conquer as follows: Since U is star-shaped its boundary ∂U is a sequence of circular arcs with vertices in between. We maintain the rays from p to these vertices sorted clockwise, say. Then it is easy to merge two boundaries of unions of $n/2$ disks into one of n disks.
3. Sort all points of Q around p in a clockwise fashion and merge them with the vertices of ∂U.
4. Perform a circular sweep around p with the points of Q and the vertices of ∂U as event points. During the sweep, compare each point $q \in Q$ encountered with the intersection point of the ray with the current circular arc of ∂U. Thus, it can be determined whether q is also in U. If so, mark the entry (p, q) in the free space matrix as "free".

Correctness. For the correctness of the algorithm, observe that for any pair (p, q) chosen in step 4 it must be true that q lies in one of the disks which contain p, and vice versa.

Runtime. One execution of step 1 takes time $O(m + n)$. In step 2, the complexity of ∂U is $O(n)$, see, e.g., [1]. The merging can be done in linear time, so the divide-and-conquer algorithm takes time $O((m + n) \log(m + n))$. Step 3 takes time $O(m \log m)$ for the sorting and $O(m + n)$ for the merging. Step 4 takes linear time. Since these steps are carried out for each point $p \in P$ the total runtime for setting up the free space matrix is $O(n(m + n) \log(m + n))$. Since the roles of P and Q can be exchanged we can achieve $O(\min(m, n)(m + n) \log(m + n))$ which is $O(mn \log(mn))$.

Output a middle curve. If in addition to a yes-answer for the decision problem also a covering sequence itself is wanted, each circular segment of ∂U should be labeled with the center point of its circle. This label is also entered into the free space matrix so that the sequence of labels of a monotone path gives a feasible unordered sequence for the middle curve.

Optimization problem. Solving the optimization problem can again be done by a binary search on the set of distances between pairs of points from $P \cup Q$ involving in each step the algorithm for the decision problem. This results in a $O(mn \log^2 mn)$ runtime.

Several curves. The decision algorithm can be extended to k curves $P^1, ..., P^k$. Then, having the outer loop for all points $p \in P^1$, say, in step 4 we determine which points $p_2 \in P^2, \ldots, p_k \in P^k$ lie inside U, as well. For all combinations $p, p_2, ..., p_k$ the corresponding entries in the k-dimensional free space matrix are marked as free. The runtime is $O(n_1 N \log N + M)$ where $N = \sum_{i=1}^{k} n_i$ and $M = \prod_{i=1}^{k} n_i$, which is only a minor improvement over the brute force algorithm with run time $O(N(N + M))$.

3 Dynamic programming for the ordered case

Now we present a dynamic programming algorithm for computing an ordered middle curve. As input we assume two sequences P, Q and we search for an ordered middle curve R. Let us denote by $P_i, 1 \leq i \leq n$, the "prefix" $(p_1, ..., p_i)$ of a sequence $P = (p_1, ..., p_n)$. P_0 is defined as the empty sequence.

Our dynamic programming algorithm operates with four-dimensional Boolean arrays of the form $X[i, j, k, l], 0 \leq i, k \leq n, 0 \leq j, l \leq m$, where $X[i, j, k, l]$ is `true` iff there exists an ordered sequence R from points in $P_i \cup Q_j$ with

$$\max(d_F(R, P_k), d_F(R, Q_l)) \leq \varepsilon.$$

We say in this case that R *covers* P_k and Q_l. Clearly, the decision problem has a positive answer iff $X[n, m, n, m]$ (or any $X[i, j, n, m]$) is true.

In order to determine the value of some $X[i, j, k, l]$ from entries of X with lower indices, we need more information, particularly, whether there is a covering sequence R in which the points p_i and q_j occur, and if they do, whether they occur in the interior or at the end of the sequence. To this end, the array X is the componentwise disjunction of seven Boolean arrays

$$X = A \vee B \vee C \vee D \vee E \vee F \vee G$$

with the meanings that a sequence R covering P_k and Q_l exists with the following properties, respectively:

$A[i,j,k,l]$: R contains neither p_i nor q_j.

$B[i,j,k,l]$: R contains p_i in its interior but does not contain q_j.

$C[i,j,k,l]$: R ends in p_i but does not contain q_j.

$D[i,j,k,l]$: R contains q_j in its interior but does not contain p_i.

$E[i,j,k,l]$: R ends in q_j but does not contain p_i.

$F[i,j,k,l]$: R contains q_j in its interior and ends in p_i.

$G[i,j,k,l]$: R contains p_i in its interior and ends in q_j.

Observe that R cannot contain both, p_i and q_j, in its interior (i.e. not at the end).

The entries of the arrays can be initialized or computed from entries with lower indices because of the following identities, which hold for each index $i,j,k,l \geq 1$, if that index minus 1 occurs in the formula and for all indices ≥ 0 otherwise.

$$\begin{array}{lcl} A[0,0,0,0] & = & \texttt{true} \\ A[0,0,k,l] & = & \texttt{false for } k \geq 1 \text{ or } l \geq 1 \\ A[i,0,k,l] & = & X[i-1,0,k,l] \\ A[0,j,k,l] & = & X[0,j-1,k,l] \\ A[i,j,k,l] & = & X[i-1,j-1,k,l] \end{array}$$

$$\begin{array}{lcl} B[i,0,k,l] & = & B[0,j,k,l] \;=\; \texttt{false} \\ B[i,j,k,l] & = & G[i,j-1,k,l] \vee B[i,j-1,k,l] \end{array}$$

The first equality is correct, since p_i must be at the end of R if no points from Q are available. In the second equality, $G[i,j-1,k,l]$ accounts for the case that R contains q_{j-1} (which then must be at the end) and $B[i,j-1,k,l]$ for the case that it does not.

In the following, let $cl(p,q)$ for points p and q denote the truth value for $\|p-q\| \leq \varepsilon$. These can be determined for all pairs of points in $P \cup Q$ by preprocessing. The following equalities for $C[i,j,k,l]$ are obtained by case distinction whether the final point p_i in the sequence R covers only p_k and q_l or also other points occurring previously in the sequences P_k and Q_l, respectively.

$$\begin{array}{lcl} C[i,j,0,l] & = & C[i,j,k,0] \;=\; C[0,j,k,l] \;=\; \texttt{false} \\ C[i,j,k,l] & = & cl(p_i,p_k) \;\wedge\; cl(p_i,q_l) \;\wedge \\ & & (A[i,j,k-1,l-1] \vee C[i,j,k-1,l-1] \\ & & \vee\, C[i,j,k-1,l] \vee C[i,j,k,l-1]) \end{array}$$

The entries of D and E can be determined analogously to the ones of B and C with the roles of p_i and q_j exchanged. The identities of F have similar explanations as the ones of C:

$$\begin{array}{lcl} F[0,j,k,l] & = & F[i,0,k,l] \;=\; F[i,j,0,l] \\ & = & F[i,j,k,0] \;=\; \texttt{false} \\ F[i,j,k,l] & = & cl(p_i,p_k) \wedge cl(p_i,q_l) \wedge \\ & & (D[i,j,k-1,l-1] \vee E[i,j,k-1,l-1] \\ & & \vee\, F[i,j,k-1,l] \vee F[i,j,k,l-1]) \end{array}$$

The entries of G can be determined analogously to the ones of F with the roles of p_i and q_j exchanged.

Runtime. The dynamic programm runs in time $O(n^2m^2)$ which is the size of each of the eight arrays.

Output a middle curve. Not only the existence of a covering sequence R, but R itself can be computed by setting a pointer for each array entry of the form $Y[i,j,k,l]$, which is set to true, to the 4-tupel(s) of indices at the right hand side of an equality that has made it true. Note that there can be an exponential number of valid middle curves.

Optimization problem. The value of $\max(d_F(R,P), d_F(R,Q))$ must be one of the distances between two points in $P \cup Q$. Therefore, the optimization problem can be solved by determining these distances, sorting them, and finding the correct value by binary search, invoking in each step the decision algorithm with the current value of ε. Altogether, this takes time $O(n^2m^2 \log(n+m))$.

Several Curves. The decision (and optimization) algorithm can be generalized to k sequences $P^1, ..., P^k$. The runtime in this case is $O(n_1^2 ... n_k^2)$ for constant k (but the number of arrays is $2^{k-1}(k+2)-1$).

4 Algorithm for the Restricted Case

Now the reparameterizations for minimizing $\max(d_F(R,P), d_F(R,Q))$ are restricted to map every vertex of R to itself in the input curve it originated from. This case allows for a more efficient dynamic program.

For this, we define arrays akin to Section 3. Let $X[i,j], 0 \leq i \leq n, 0 \leq j \leq m$, be `true` iff there exists an ordered sequence R from points in $P_i \cup Q_j$ with

$$\max(d_F(R,P_i), d_F(R,Q_j)) \leq \varepsilon,$$

with the restriction that any vertex of R is mapped to itself in the input curve it originated from. We say in this case that R *restrictively covers* P_i and Q_j. Clearly, the decision problem has a positive answer iff $X[n,m]$ is true.

Akin to Section 3 we can write X as a disjunction of three Boolean arrays

$$X = A \vee C \vee E$$

with the meanings that a sequence R covering P_i and Q_j exists with the following properties[1], respectively:

$A[i,j]$: R contains neither p_i nor q_j

$C[i,j]$: R ends in p_i (and may or may not contain q_j)

$E[i,j]$: R ends in q_j (and may or may not contain p_i)

[1] note that C here combines C and F in Section 3, and E combines E and G in Section 3

First we observe that

$A[i,j] \Leftrightarrow \exists\ i' < i$ and $j' < j$ such that $(C[i',j'] \wedge (i,j) \in U_P(i',j')) \vee (E[i',j'] \wedge (i,j) \in U_Q(i',j'))$

$C[i,j] \Leftrightarrow cl(p_i,q_j) \wedge$ there exist $i' < i$ and $j' < j$ such that $X[i',j'] \wedge (i',j') \in \hat{L}_P(i,j)$

$E[i,j] \Leftrightarrow cl(p_i,q_j) \wedge$ there exist $i' < i$ and $j' < j$ such that $X[i',j'] \wedge (i',j') \in \hat{L}_Q(i,j)$

Here, the *upper right* wedge $U_P(i',j')$ and the *lower left wedge* $L_P(i',j')$ represent subsets of point pairs (p_i,q_j) for which p_i and q_j are both close to $p_{i'}$. The upper right wedge consists of the *connected* set of such close point index pairs (i,j) for which $i' \le i$, $j' \le j$, and the set contains (i',j'). The lower left wedge consists of the *connected* set of such close point index pairs (i,j) for which $i \le i'$, $j \le j'$, and the point set contains (i',j').

Finally, we define the *extended lower left wedge* $\hat{L}_P(i',j')$ which, in addition to all points in the lower left wedge $L_P(i',j')$ also contains the points (i,j) immediately to the left or below, i.e., for which $(i+1,j)$, $(i,j+1)$, or $(i+1,j+1)$ is contained in $L_P(i',j')$. The definition of $U_Q(i',j'), L_Q(i',j'), \hat{L}_Q(i',j')$ is analogous, consisting of point pairs (p_i,q_j) for which p_i and q_j are both close to $q_{i'}$.

We compute X by incrementally adding true points using an enhanced bottom-up dynamic programming. In addition to storing the $(m+1) \times (n+1)$-array X, we also store the upper envelope $\bar{X}$ of all true points in X as a 1D array indexed by i. This will allow us to efficiently add reachable points to X. More specifically, we define $\bar{X}[i] = \max\{j \mid X[i,j] = true\}$. Note that X as well as $\bar{X}$ change during the dynamic programming, as more and more true points get added to X.

First, initialize all $X[i,j]$ to `false`, except for $X[0,0]$ which is set to `true`. Initialize $\bar{X}[0] = 0$, and $\bar{X}[i] = -1$ for all $i > 0$.

Then, for $i = 1$ to m, and for $j = 1$ to n, compute $X[i,j]$ (and update $\bar{X}$) as follows:

- If $X[i,j] \wedge cl(p_i,q_j)$: Add $U_P(i,j)$ and $U_Q(i,j)$ to X, together with a pointer to (i,j) that is labeled P or Q accordingly. An upper wedge is added to X by locating it in $\bar{X}$, updating the points in X that are above $\bar{X}$ by setting them to `true`, and finally updating $\bar{X}$. We refer to this as *updating X and $\bar{X}$ with the wedge*. This takes time proportional to the number of points updated.

- If $\neg X[i,j] \wedge cl(p_i,q_j)$: If $\hat{L}_P(i,j)$ intersects $\bar{X}$, update X and $\bar{X}$ with $L_P(i,j)$ and $U_P(i,j)$, and if $\hat{L}_Q(i,j)$ intersects $\bar{X}$, update X and $\bar{X}$ with $L_Q(i,j)$ and $U_Q(i,j)$. Also update pointers labeled with P or Q accordingly. The check can be done in constant time, and the update takes time proportional to the number of new points updated.

Correctness. For the correctness of the algorithm observe that if $X[i,j]$ holds because of $A[i,j]$, then it is marked when the last point of a covering is processed. If $X[i,j]$ holds by $C[i,j]$ or $E[i,j]$, then this is handled in the $\neg X[i,j] \wedge cl(p_i,q_j)$ case of the algorithm.

Runtime. By storing $\bar{X}$ in a binary search tree the algorithm runs in time $O(mn \log(\min(m,n)))$. For this, store $\bar{X}$ in a binary search tree sorted on i and augmented by the minimum value $\bar{X}[i]$ in a subtree rooted at a node. A rectangle with corners (i,j) and (u,r) can be queried by following the two paths to i and u. In between those paths process all subtrees with minimum smaller than r. Updating the values for $\bar{X}[i]$ and the minimum of these takes logarithmic time as well. Thus, it takes at most logarithmic time both to mark and to process an entry of X.

Several Curves. For $k > 2$ curves the algorithm works the same with a $k-1$ dimensional range tree for $\bar{X}$, and runtime $O(n^k \log^{k-1} n)$.

Acknowledgments. This work was initiated at the 17th Korean Workshop on Computational Geometry. We thank the organizers and all participants for the stimulating atmosphere. In particular we thank Fabian Stehn and Wolfgang Mulzer for discussing this paper.

References

[1] Pankaj K. Agarwal, János Pach, and Micha Sharir. State of the union (of geometric objects). In Jacob E. Goodman, Janos Pach, and Richard Pollack, editors, *Surveys on Discrete and Computational Geometry, Twenty Years Later*, volume 453 of *Contemporary Mathematics*, pages 9–48. ams, 2006.

[2] Helmut Alt and Michael Godau. Computing the Fréchet distance between two polygonal curves. *Int. J. Comput. Geometry Appl.*, 5:75–91, 1995.

[3] Kevin Buchin, Maike Buchin, Marc van Kreveld, Maarten Löffler, Rodrigo I. Silveira, Carola Wenk, and Lionov Wiratma. Median trajectories. *Algorithmica*, 66(3):595–614, 2013.

[4] Thomas Eiter and Heikki Mannila. Computing discrete Fréchet distance. Technical report, Technische Universität Wien, 1994.

[5] Sariel Har-Peled and Benjamin Raichel. The Fréchet distance revisited and extended. *ACM Trans. Algorithms*, 10(1):3:1–3:22, January 2014.