Middle Curves Based on Discrete Fréchet Distance *

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

July 29, 2018

Abstract

Given a set of polygonal curves, we present algorithms for computing a *middle curve* that serves as a representative for the entire set of curves. We require that the middle curve consists of vertices of the input curves and that it minimizes the maximum discrete Fréchet distance to all input curves. We consider three different variants of a middle curve depending on in which order vertices of the input curves may occur on the middle curve, and provide algorithms for computing each variant.

1 Introduction

Sequential point data, such as time series and trajectories, are ever increasing due to technological advances, and the analysis of these data calls for efficient algorithms. An important analysis task is to find a "representative" or "middle" curve for a set of similar curves. For instance, this could be the route of a group of people or animals traveling together. Or it could be a representation of a handwritten letter for a class of similar handwritten letters. Such a middle curve typically provides a concise representation of the data, which is useful for data analysis and for reducing the size of the data.

Since sampled locations are more reliable than positions interpolated in between those, we seek a middle curve consisting only of sampled point locations. The middle curve should then be as close as possible to the individual curves, hence we ask for it to minimize the maximum discrete Fréchet distance d_F to any of these. The Fréchet distance [1] and the discrete Fréchet distance [6] are well-known distance measures, which have been successfully used in analyzing handwritten characters [8] and trajectories [2, 10].

For simplicity, we restrict our definitions to sets of two polygonal curves P and Q, as the generalization to $k \geq 2$ such curves is straightforward. Given P and Q, we consider polygonal curves R with vertices from $P \cup Q$ where we assume that each vertex of R uniquely corresponds

*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 grants CCF-1301911 and CCF-1618469. Work by Ahn and Oh was supported by the MSIT(Ministry of Science and ICT), Korea, under the SW Starlab support program(IITP–2017–0–00905) supervised by the IITP(Institute for Information & communications Technology Promotion) and the NRF grant 2011-0030044 (SRC-GAIA) funded by the government of Korea.

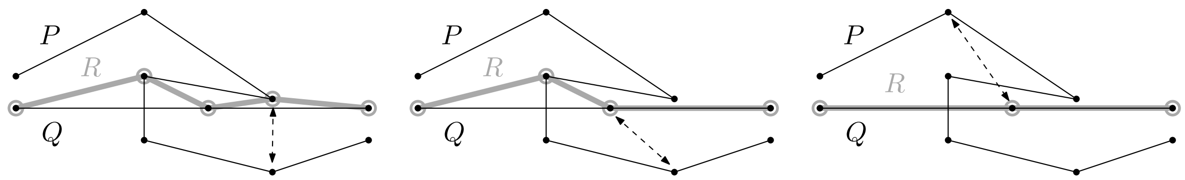
†Pohang University of Science and Technology, Korea. Email: `heekap@postech.ac.kr`

‡Free University of Berlin, Germany. Email: `{alt, scharf}@mi.fu-berlin.de`

§Technical University Dortmund, Germany. Email: `maike.buchin@tu-dortmund.de`

¶Max Planck Institute for Informatics, Germany. Email: `eoh@mpi-inf.mpg.de`

||Tulane University, United States. Email: `cwenk@tulane.edu`



(a) unordered (b) ordered (c) restricted

Figure 1: Illustration of the three different cases. The curve R is a middle curve for each case. The two-way arrow which points to a vertex in $P \cup Q$ and a vertex in R indicates a mapping between two vertices realizing the discrete Fréchet distance.

to a vertex of P or Q. That is, we identify any such vertex by its index in P or Q and not just by the geometric position of the corresponding point.

R is called *ordered* if any two elements of P occurring in R have the same order as in P, likewise with elements from Q. If R minimizes $\max\{d_F(R,P),\ d_F(R,Q)\}$, it is called a (unordered) *middle curve* of P and Q. If R is ordered and minimizes this expression for all ordered curves it is called an *ordered middle curve* of P and Q.

We obtain a third variant of the definition of middle curve, called *restricted middle curve*, by restricting the set of feasible matchings (see Section 2) in the definition of the discrete Fréchet distance. More precisely, R should be ordered and only matchings are allowed where elements of R are matched to their corresponding elements in P or Q. R then should be closest to P and Q among all ordered sequences with respect to this distance measure. Since vertices of R originate from P or Q, this seems a natural restriction.

Figure 1 illustrates the three cases we consider: unordered, ordered, and restricted middle curves. In this example, each case results in a different middle curve, and the associated distance increases as we add more restrictions. Note that from unordered to ordered we limit the middle curves we consider, whereas from ordered to restricted we limit the matchings we consider. Respecting the order of the input curves seems to be a natural requirement (ordered middle curve). Furthermore it seems intuitive that a vertex should be matched to itself on the middle curve (restricted middle curve). Among the different algorithms we present for the various cases, the most efficient algorithms are for computing an unordered middle curve in the Euclidean plane and (slightly less efficient) for computing a restricted middle curve.

Related work. The problem of finding a curve that represents a set of curves has been studied in the literature [4, 7, 9]. While there are different definitions of such a representative curve, none of them requires the representative curve to use vertices of the input curves. Buchin et al. [4] and van Kreveld et al. [9] both require the representative curve to use parts of the input edges. Buchin et al. aim for the curve to always “stay in the middle” in the sense of a median and give an $O(k^2n^2)$-time algorithm, where k is the number of given curves and n is the number of vertices in each curve. van Kreveld et al. require the representative curve to be as close as possible to all trajectories at any time, allowing small jumps between different trajectories, and give an $O(k^3n)$-time algorithm. Note that neither of these approaches makes use of the Fréchet distance or its variants. Using neither input vertices nor input edges, Har-Peled and Raichel [7] show that a curve minimizing the Fréchet distance to k input curves can be computed in $O(n^k)$ time in the k-dimensional free space using the radius of the smallest enclosing disk as “distance”.

Our Results. We present algorithms for computing a middle curve that minimizes the discrete Fréchet distance to k input curves for $k \geq 2$ each of size at most n in three variants:

1. **Ordered case:** $O(n^{2k})$-time algorithm for computing an ordered middle curve.

2. **Restricted case:** $O(n^k \log^k n)$-time algorithm for computing a restricted middle curve.

3. **Unordered case:** $O(n^k \log n)$-time algorithm for computing an unordered middle curve.

In the following sections, we present the algorithms for these three cases. For the ordered case and the restricted case, the algorithm works for any metric. For the unordered case, the algorithm works only for the Euclidean metric while we can modify our algorithm to work for any metric with running time of $O(n^{k+1})$.

We also distinguish two variants of the problem depending on whether multiple occurrences of vertices on R are allowed or not. The algorithms for the restricted and the unordered cases allow vertices to occur multiple times. In the ordered case, our algorithms can handle both variants.

Instead of minimizing the distance to the input curves, we may want to minimize the size of a middle curve for a fixed distance. We give algorithms for this variant as well. However in the end we discuss that middle curves may have high complexity.

2 Preliminaries

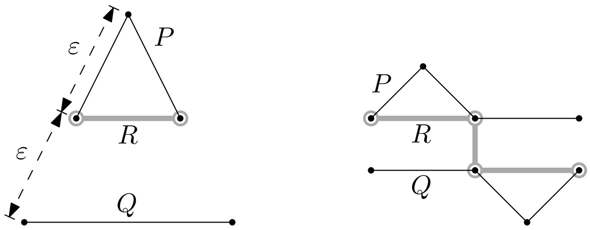
Let $P = (p_1, \ldots, p_n)$ and $Q = (q_1, \ldots, q_m)$ be two polygonal curves, represented as point sequences in $\mathbb{R}^{\mathbf{d}}$. And let $d(.,.)$ denote a metric used to measure point-wise distances. The discrete Fréchet distance, denoted by $d_F(P, Q)$, is defined as follows: A *matching*[1] is a sequence of pairs $(p, q) \in P \times Q$ such that (1) the sequence begins at (p_1, q_1) and ends at (p_n, q_m), and (2) a pair (p_i, q_j) in the sequence is followed only by one of (p_{i+1}, q_j), (p_i, q_{j+1}), or (p_{i+1}, q_{j+1}). The value of a matching is the maximum distance of p and q over all pairs (p, q) in the matching. Then $d_F(P, Q)$ is the minimum value over all possible matchings. If P and Q are empty, then we define $d_F(P, Q) = 0$, and if either P or Q is empty then $d_F(P, Q) = \infty$.

As mentioned in the introduction, given two polygonal curves represented as point sequences $P = (p_1, \ldots, p_n)$ and $Q = (q_1, \ldots, q_m)$, a (unordered) *middle curve* of P and Q is a sequence R of elements from $P \cup Q$ which minimizes $\max\{d_F(R, P),\ d_F(R, Q)\}$. Notice, that an element of R is identified by whether it originates from P or Q and by the index in that sequence. This is necessary when we consider the other variants of middle curve as defined in the introduction. We call R an *ordered middle curve* if elements of R respect the order given by the input curves and $\max\{d_F(R, P),\ d_F(R, Q)\}$ is the minimum among all such curves. We also consider the *restricted* variant of middle curves, where elements of R need to be matched to themselves in the curve they are taken from.

2-Approximation. A simple observation is that any of the input curves is a 2-approximate middle curve, i.e., the associated discrete Fréchet distance is at most double that of an optimal middle curve. Any input curve gives a restricted middle curve, and hence this holds for all three variants, i.e., unordered, ordered, and restricted.

The 2-approximation follows by the triangle inequality. In fact, let P and Q be two arbitrary input curves. For any middle curve R realizing the minimum distance, say $d_{\min}$, let (p_i, r_k) and (q_j, r_k) denote pairs in optimal matchings of R with P and Q, respectively. Then, we have $d(p_i, q_j) \leq d(p_i, r_k) + d(r_k, q_j) \leq 2d_{\min}$ by the triangle inequality, where $d(p, q)$ denotes the

[1] Note that this is not a one-to-one matching, and for this reason has also been called a coupling.



(a) (b)

Figure 2: (a) The 2-approximation is tight. (b) The middle curve may need to consist of vertices from both curves.

distance between p and q of the underlying metric. As can easily be seen, this implies that there is a matching of P and Q whose distance is at most $2d_{\min}$. Thus, we have a 2-approximation in constant time (not counting the time to output the vertices of the approximate middle curve). Note that the 2-approximation is tight, as the example in Figure 2(a) shows. We observe also that for a middle curve we may need to choose a subset of vertices from both curves, as in Figure 2(b).

3 Algorithms for the Ordered Case

In this section we present a dynamic programming algorithm for computing an ordered middle curve R. We first give a decision algorithm for two input curves P and Q, and we allow the same vertex to occur at most once on R. Later we show how to generalize the algorithm to more than two input curves, to allow vertices to occur multiple times in R, and how to solve the two optimization variants.

3.1 Decision Problem for Two Curves

Let $P = (p_1, \ldots, p_n)$ and $Q = (q_1, \ldots, q_m)$ be two input curves, and let $\varepsilon \geq 0$ be given. In this section we allow the same vertex to occur at most once on R. Let P_i denote the *prefix* $(p_1, \ldots, p_i)$ of P for $1 \leq i \leq n$, and let Q_j denote the prefix $(q_1, \ldots, q_j)$ of Q for $1 \leq j \leq m$. We use P_0 and Q_0 to denote an empty subsequence of P and Q, respectively.

The dynamic programming algorithm operates with four-dimensional Boolean arrays of the form $X[i, j, k, l]$ for $0 \leq i \leq k \leq n$ and $0 \leq j \leq l \leq m$, where $X[i, j, k, l]$ is `true` if and only if there exists an ordered sequence R from points in $P_i \cup Q_j$ for a fixed $\varepsilon \geq 0$ such that

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

We say in this case that R *covers* P_k and Q_l. Clearly, the decision problem has a positive answer if and only if $X[i, j, n, m]$ is true for some i and j.

In order to determine the values of $X[i, j, k, l]$ with $0 \leq i \leq k \leq n$ and $0 \leq j \leq l \leq m$, 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, we can represent the array X as the component-wise disjunction of seven Boolean arrays

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

The entries of the Boolean arrays defined below, indicate whether an ordered sequence R from points in $P_i \cup Q_j, 0 \leq i \leq n, 0 \leq j \leq m$ 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.

If $i = 0$ or $j = 0$, the described properties involve the "nonexisting points" p_0 or q_0 which should be interpreted as not being contained in any curve. Also, it might be the case that i, j, k, or l are 0 which means that the corresponding sequences are empty. In general, observe that R cannot contain both p_i and q_j in its interior. See Figure 3 for an illustration of the seven different cases that can occur.

Our dynamic programming algorithm is based on the recursive identities for the Boolean arrays given in the following paragraphs. Each identity holds only if all index ranges of the arrays in the formulas are nonnegative.

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

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

As easily can be verified, each of these identities holds with the meaning given to the Boolean arrays previously. For example, the first two equalities for A hold because an empty curve covers an empty sequence, but not any other sequence, and the following equalities for A are straightforward. The first line of equalities for B holds because p_i must be at the end of R if no points from Q are available, and there is no point p_0 which a middle curve could contain. In the last equality for B, the entry $G[i, j-1, k, l]$ 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.

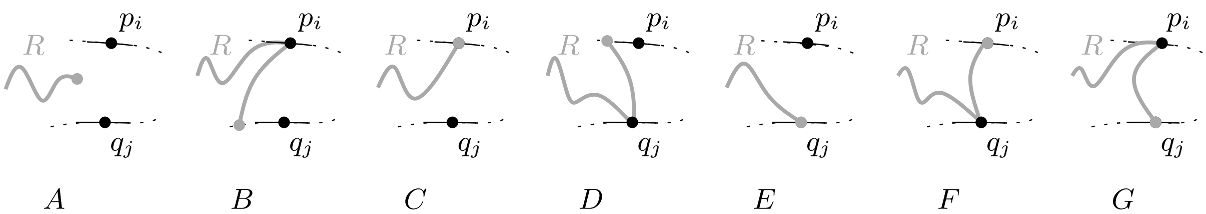


Figure 3: Illustration of cases in the dynamic programming.

In the following, let $cl(p,q) = \texttt{true}$ if and only if $d(p,q) \leq \varepsilon$, for two points p and q. The following identities hold for C:

$$
\begin{aligned}
C[i,j,0,l] &= C[i,j,k,0] = \ C[0,j,k,l] = \ \texttt{false} \qquad\qquad \text{and otherwise,}\\
C[i,j,k,l] &= cl(p_i,p_k) \ \wedge \ cl(p_i,q_l) \ \wedge \\
&\qquad\qquad (\ A[i,j,k-1,l-1] \vee A[i,j,k-1,l] \vee A[i,j,k,l-1] \vee \\
&\qquad\qquad\ \ C[i,j,k-1,l-1] \vee C[i,j,k-1,l] \vee C[i,j,k,l-1] \quad)
\end{aligned}
$$

The equalities in the first line hold because only an empty curve can cover an empty sequence, and because a middle curve cannot end in the nonexisting p_0. The equality in the second line models that the final point p_i in R can cover p_k and q_l only, or it can also cover additional points that occur earlier in the sequences P_k and Q_l.

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 for F have similar explanations as the ones for C:

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

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

The dynamic programming algorithm takes $O(n^2m^2)$ time and space to decide if an ordered middle curve of two curves of size n and m exists for a fixed distance. While filling the dynamic programming matrices we can compute an additional pointer array $Y[i,j,k,l]$ that, for each `true` assignment in one of the equalities, stores a pointer to one of the 4-tuples of indices on the right hand side of the equality that made the assignment `true`. A covering sequence R can then be computed by backtracking these pointers. Note that there can be an exponential number of valid middle curves (e.g., if all points are within distance ε of each other).

3.2 Optimization Problems

We can solve the optimization problem of minimizing ε by adapting the dynamic programming approach to compute the smallest value for which a covering middle curve exists.

Instead of storing truth values, X stores the minimum value of the seven arrays, A to G. Array entries are initialized to $0|\infty$ instead of `true|false`, any $cl(p,q)$ in the formulas is replaced by the distance $d(p,q)$, $\vee$ becomes min, and $\wedge$ becomes max. The running time for computing the minimum ε for which a middle curve exists is the same as for the decision problem described in Section 3.1.

Similarly, if we want to minimize the size of an ordered middle curve for a fixed distance ε, we can adapt the dynamic program to store the smallest size of a middle curve in the following way:

Again, `true|false` are replaced by $0|\infty$, $\vee$ becomes min, and the recursion for C is modified as follows:

$$
C[i,j,k,l] = \infty \qquad \text{if } \neg(cl(p_i,p_k) \wedge cl(p_i,q_l)) \text{ and, otherwise,}
$$

$$
\begin{aligned}
C[i,j,k,l] = \min(\ & A[i,j,k-1,l-1]+1,\ A[i,j,k-1,l]+1,\ A[i,j,k,l-1]+1,\\
& C[i,j,k-1,l-1],\ C[i,j,k-1,l],\ C[i,j,k,l-1])
\end{aligned}
$$

and the one for F becomes:

$$F[i,j,k,l] = \infty \qquad \text{if } \neg(cl(p_i,p_k) \wedge cl(p_i,q_l)) \text{ and, otherwise,}$$

$$\begin{aligned} F[i,j,k,l] = \min(\; & D[i,j,k-1,l-1]+1,\ D[i,j,k-1,l]+1,\ D[i,j,k,l-1]+1, \\ & E[i,j,k-1,l-1]+1,\ E[i,j,k-1,l]+1,\ E[i,j,k,l-1]+1, \\ & F[i,j,k-1,l-1],\ F[i,j,k-1,l],\ F[i,j,k,l-1]) \end{aligned}$$

Again, the entries of E and G can be computed analogously to the ones of C and F with the roles of p_i and q_j exchanged.

3.3 Generalization to Multiple Curves and Multiple Vertex Occurrences

The decision and optimization algorithms can be generalized to k sequences $P^1,\ldots,P^k$ of sizes $n_1,\ldots,n_k$, respectively. The corresponding arrays then have $2k$ indices $[i_1,...,i_k,j_1,...,j_k]$, $1 \le i_1,j_1 \le n_1,...,1 \le i_k,j_k \le n_k$, describing the covering of the partial sequences (prefixes) $P^1_{j_1},...,P^k_{j_k}$ by a middle curve R using points from $P^1_{i_1} \cup ... \cup P^k_{i_k}$.

As in the case $k=2$, we obtain different arrays depending on which points $p^1_{i_1},...,p^k_{i_k}$ are at the end, in the interior, or not contained in R. Since there are $k2^{k-1}$ possibilities to put one of these points at the end and others in the interior, and 2^k possibilities to put a subset of $\{p^1_{i_1},...,p^k_{i_k}\}$ in the interior and not using the remaining ones, and the possibility of having all of them in the interior is excluded, there must be $k2^{k-1}+2^k-1$ arrays reflecting all these possibilities. (Which is, in fact, 7 for $k=2$).

For any constant k, there is a constant number of arrays each of which has $n_1^2 \cdots n_k^2$ entries. The recursive formulas to compute the entries of the arrays which, for simplicity, we do not explain for the general case, can be derived in a way analogous to the ones in the case $k=2$. They have constant size for constant k, and therefore the runtime of the dynamic programming algorithm is $O(n_1^2 \cdots n_k^2)$.

The dynamic programming algorithm can also be modified to allow multiple occurrences of points on R, which requires distinguishing slightly more cases than before: Whether a point appears at the end only, both at the end and in the interior, in the interior only, or not at all in R. This results in $2^k(k+1)-1$ arrays: now all k points may appear in the interior or not, any of the k or no point appears at the end, but not all k points can appear in the interior only.

We summarize the results of this section:

Theorem 1. *For $k \ge 2$ curves of size at most n each, we can compute an ordered middle curve in $O(n^{2k})$ time using $O(n^{2k})$ space. For fixed $\varepsilon > 0$, an ordered middle curve with minimum complexity and distance at most ε to the input curves can be computed in the same time.*

4 Algorithms for the Restricted Case

Now we consider the case where the matchings realizing the discrete Fréchet distance 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 programming algorithm.

4.1 Decision Problem for Two Curves

Dynamic Programming Formulation. Let $P=(p_1,\ldots,p_n)$ and $Q=(q_1,\ldots,q_m)$ be two input curves with $m \le n$, and let $\varepsilon \ge 0$. We define arrays similar to the ones in Section 3. Let

$X[i,j]$ be `true` for $0 \le i \le n, 0 \le j \le m$ if and only if there exists an ordered sequence R from points in $P_i \cup Q_j$ with

$$\max\{d_F(R, P_i), d_F(R, Q_j)\} \le \varepsilon,$$

with the restriction that there exist matchings that map every vertex of R 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 if and only if $X[n,m]$ is `true`.

Similar to the one in Section 3 we can write X as a disjunction of three Boolean arrays

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

For each array defined below, a sequence R from points in $P_i \cup Q_j$ restrictively covering P_i and Q_j exists with the following properties, respectively:

$A'[i,j]$: R ends in neither p_i nor q_j (but may contain one of them in its interior).

$C'[i,j]$: R ends in p_i (and may contain q_j in its interior).

$E'[i,j]$: R ends in q_j (and may contain p_i in its interior).

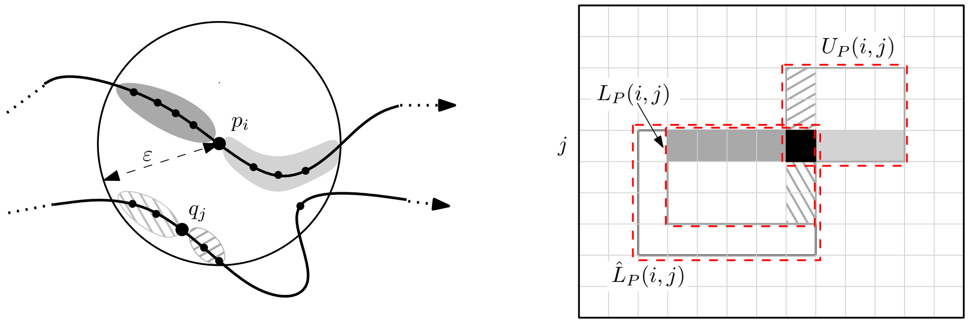
In contrast to the one in Section 3, we now only distinguish the cases by the last point of R. Hence, we only distinguish three cases. (In comparison to the ordered case, A' combines A, B, D, and C' combines C, F, and E' combines E, G). However, we will only explicitly compute X and not A', C', E'.

We compute X incrementally using dynamic programming, iterating over all (i,j) in increasing order. For fixed (i,j) we consider adding p_i or q_j to a middle curve R from $P_i \cup Q_j$, where p_i and q_j are matched to each other. That is, we consider adding say p_i to R, where it is matched to itself on P and to the point q_j on Q. We iterate over all (i,j) maintaining in the table X the coverage of P and Q by a restricted middle curve from $P_i \cup Q_j$.

For this, we initialize $X[0,0]$ to `true` and $X[i,0]$ and $X[0,j]$ to `false` for $i,j > 0$. Then we incrementally process all entries (i,j) for $i,j > 0$. We use the following order (which we will need later when we introduce $\bar{X}$): For $j = 1$, we process all (i,j) incrementally from $i = 1$ to n; Then we increase the index j by 1 and process all (i,j) incrementally from $i = 1$ to n, and we repeat this up to $j = m$.

When processing (i,j), we need to first check, whether p_i and q_j can be matched to each other, i.e., they have distance $\le \varepsilon$. If they do, we then need to check if there exists a middle curve from $P_i \cup Q_j$ that covers up to here, or whose coverage can be extended up to here after adding p_i or q_j. If this is the case, we consider how the coverage extends by adding p_i or q_j or both. We maintain this information in the table X as we iterate over (i,j). To make this more explicit, we introduce the notions of upper and lower wedges in the table X, which reflect the coverage on P and Q when adding a point p_i or q_j, which are matched to each other.

Assume p_i is matched to q_j. We use the *upper wedge* $U_P(i,j)$ to describe the resulting coverage of P and Q when adding p_i to R. Specifically, $U_P(i,j)$ denotes the set of index pairs (i',j') such that $d(p_{i''}, p_i) \le \varepsilon$ and $d(q_{j''}, p_i) \le \varepsilon$ for all $i \le i'' \le i'$ and $j \le j'' \le j'$. That is, $U_P(i,j)$ consists of consecutive index pairs $(i',j') \ge (i,j)$ that are covered by p_i. The *lower wedge* $L_P(i,j)$ denotes the set of index pairs (i',j') such that $d(p_{i''}, p_i) \le \varepsilon$ and $d(q_{j''}, p_i) \le \varepsilon$ for all $i' \le i'' \le i$ and $j' \le j'' \le j$. Furthermore, we define the *extended lower wedge* $\hat{L}_P(i,j)$ which, in addition to all index pairs in the lower wedge $L_P(i,j)$ also contains (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 wedges $U_Q[i,j]$, $L_Q[i,j]$, and $\hat{L}_Q[i,j]$ are defined analogously, consisting of index pairs (i',j') for which $p_{i'}$ and $q_{j'}$ are both close to q_j, that is, $cl(p_{i'}, q_j) = cl(q_{j'}, q_j) = $ `true`. Figure 4 illustrates these wedges for a pair (i,j).



(a) Curves P and Q (b) Table X

Figure 4: (a) Points of P and Q that are at distance at most ε from p_i. (b) The upper wedge $U_P(i,j)$, the lower wedge $L_P(i,j)$, and the expended lower edge $\hat{L}_P(i,j)$ at (i,j) on P.

Using this terminology we observe for $i,j>0$:

$$\begin{aligned}
A'[i,j] &= (\ \exists i' < i, j' \le j : (C'[i',j'] \wedge (i,j) \in U_P(i',j'))\ ) \\
&\quad \vee (\ \exists i' \le i, j' < j : (E'[i',j'] \wedge (i,j) \in U_Q(i',j'))\ ) \\
C'[i,j] &= cl(p_i,q_j)\ \wedge (\ \exists i' \le i, j' < j : (X[i',j'] \wedge (i',j') \in \hat{L}_P(i,j))\ ) \\
E'[i,j] &= cl(p_i,q_j)\ \wedge (\ \exists i' < i, j' \le j : (X[i',j'] \wedge (i',j') \in \hat{L}_Q(i,j))\ )
\end{aligned}$$

During the dynamic programming, in order to efficiently answer queries of the form $(i,j) \in U_P(i',j')$ we maintain the upper envelope $\bar{X}$ of all true elements in X. More specifically, we define $\bar{X}[i] = \max\{j \mid X[i,j] = \texttt{true}\}$. Using $\bar{X}$, the query whether a rectangle is nonempty (i.e., contains a true point) reduces to querying whether the upper envelope intersects the rectangle. Note that X as well as $\bar{X}$ change during the dynamic programming for increasing i and j in the specified order.

Querying and Updating the Upper Envelope $\bar{X}$. We store $\bar{X}$ in an augmented balanced binary search tree sorted on i. Each leaf corresponds to an index i and stores $\bar{X}[i]$. We sometimes use an index i to denote the leaf corresponding to i. For an internal node v of the search tree, let $I(v)$ denote the set of all indices corresponding to the leaves of the subtree rooted at v. Each internal node v stores two key values $m[v]$ and $M[v]$, where $m[v]$ is the minimum of $\bar{X}[i]$ over all indices i in $I(v)$ and $M[v]$ is the maximum. Note that the node set, and hence the structure, of the tree is static. The only changes we make, is increasing the values $\bar{X}[i]$ at leaves, and hence also the augmented values at inner nodes.

We need the following two operations.

1. *Querying whether a rectangle intersects $\bar{X}$.* Given an extended lower wedge with bottom-left corner (i_B, j_B) and top-right corner (i_T, j_T), we need to check if there is an index i such that $j_B \le \bar{X}[i]$ and $i_B \le i \le i_T$.

   This can be done as follows. Consider the search paths from the root to i_B and i_T. Let u_c be the lowest common ancestor of i_B and i_T. Whenever we descend into the right child at a node v on the path from u_c to i_T, we check the maximum key value of the left child v_L of v. The set $I(v_L)$ is contained in the interval $[i_B, i_T]$. Thus, if $M[v_L] \ge j_B$, the correct answer for the query is "yes". Otherwise, we do not need to consider the subtree rooted at v_L further. Whenever we descend into the left child at a node v on the path to i_B, we check the answer for the query on the right child of v analogously. Hence we can answer the query while we traverse the two paths, which takes logarithmic time.

2. *Updating $\bar{X}$ by adding a rectangle.* Given an upper wedge whose bottom-left corner is (i_B, j_B) and top-right corner is (i_T, j_T), we want to add this to $\bar{X}$, i.e., all values in the wedge are set to `true` in X. For this, we need to update $\bar{X}[i]$ to j_T for all $i_B \le i \le i_T$ with $\bar{X}[i] < j_T$.

   We traverse the balanced binary search tree from the root as follows. Assume that we reach a node v. If j_T is at most $m[v]$ or $I(v)$ has no element lying in $[i_B, i_T]$, then we do not need to update the values stored in the leaves of the subtree rooted at v. Hence we do not traverse this subtree. If $m[v]$ is smaller than j_T and $I(v)$ has an element lying in $[i_B, i_T]$, then we need to search further in the subtree rooted at v. So, we move to both children of v.

   Finally we reach some leaf, which is updated if $j_T > \bar{X}[i]$. Then we go back to the root from those leaves and update the key values for internal nodes lying on the paths. It is easy to see that the running time of the update is $O(c \log n)$, where c is the number of indices which are updated.

We can now formulate the decision algorithm, which first precomputes all wedges and then iteratively computes the table X.

Computing all wedges. We compute the upper wedge $U_P(i,j)$ as follows: For fixed p_i, we first find the largest $k \ge i$ such that all $p_i, \ldots, p_k$ are at distance at most ε from p_i. Then we find the largest $l \ge j$ such that all $q_j, \ldots, q_l$ are at distance at most ε from p_i. This determines the upper right corner (k,l) of $U_P(i,j)$. Note that (k,l) is also the upper right corner for all $U_P(i,j')$ for $j \le j' \le l$. Hence, all upper wedges $U_P(i,j)$ for a fixed i can be computed in $O(n)$ time using two linear scans, one over P and one over Q. The wedges $U_Q(i,j)$, $L_P(i,j)$, $L_Q(i,j)$ are computed in a similar manner.

Computing the table X. First, we initialize all $X[i,j]$ to `false`, except for $X[0,0]$ which is set to `true`. Then we compute $X[i,j]$ incrementally in the specified order. In each iteration, we process (p_i, q_j) only if they can be matched to each other, i.e., if $cl(p_i, q_j) =$ `true`.

When we compute an entry $X[i,j]$, we have two cases. If $X[i,j]$ is still set to `false`, i.e., we do not know of a middle curve covering P_i and Q_j yet, we first check whether adding p_i or q_j to a covering sequence would extend the coverage to here. For this, we check if $\hat{L}_P(i,j)$ or $\hat{L}_Q(i,j)$ intersects $\bar{X}$. If $\hat{L}_P(i,j)$ intersects $\bar{X}$, then p_i can be added to a covering sequence, and we set $X[i,j] =$ `true`. Conversely, if $\hat{L}_Q(i,j)$ intersects $\bar{X}$, then q_j can be added to a covering sequence, and we do the same.

If $X[i,j]$ is `true`, then both p_i and q_j can be added to a covering sequence, and hence we add the points covered by p_i or q_j, i.e., $U_P(i,j)$ and $U_Q(i,j)$, to X and $\bar{X}$. The wedge $U_P(i,j)$ is *added to X and $\bar{X}$* as follows: We update $\bar{X}$ with $U_P(i,j)$. During the update step we can identify all pairs $(i',j') \in U_P(i,j)$ with $\neg X[i',j']$; these are all (i',j') such that i' is a leaf in $\bar{X}$ that gets updated and $\max\{j_B, \bar{X}[i']\} \le j' \le j_T$, where (i_B, j_B) is the lower left and (i_T, j_T) the upper right corner of $U_P(i,j)$. We set all $X[i',j'] =$ `true` and store a pointer from (i',j') to (i,j) that is labeled with P. Adding $U_Q(i,j)$ to X and $\bar{X}$ is done in a similar manner, but the pointers are labeled with Q. Note that the upper wedges are added to X in such a way that each $X[i,j]$ is set to `true` only once.

The algorithm can be summarized as follows.

Set $X[i,j] = \texttt{false}$ for all index pairs (i,j), except $X[0,0]$ which is set to $\texttt{true}$.
Set $\bar{X}[i] = -1$ for all indices $i > 0$, except $\bar{X}[0]$ which is set to 0.

for $j = 1$ **to** m **do**

  for $i = 1$ **to** n **do**

    if $cl(p_i, q_j) = \texttt{true}$**:**

      if $\neg X[i,j]$**:** If $\hat{L}_P(i,j)$ or $\hat{L}_Q(i,j)$ intersects $\bar{X}$, set $X[i,j]$ to $\texttt{true}$

      if $X[i,j]$**:** Add $U_P(i,j)$ and $U_Q(i,j)$ to X and $\bar{X}$, i.e. set all entries to $\texttt{true}$

For the correctness of the algorithm, observe that if $X[i,j]$ holds because of $A'[i,j]$, then it is set to $\texttt{true}$ 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 $\neg X[i,j]$ case of the algorithm.

Recall that we assume $m \leq n$. The running time for computing all wedges is $O(n^2)$ since for each point $p_i \in P$ or $q_j \in Q$, we perform a constant number of linear scans. For the main part of the dynamic programming algorithm, when we consider an index pair (i,j), we perform a query on $\bar{X}$ which takes $O(\log n)$ time, and we add one or two upper wedges to X. The update operation that is part of adding a wedge takes $O(c \log n)$ time, where c is the number of indices that are updated. However note that $\bar{X}[i]$ is updated at most m times for each index i in total, and $X[i,j]$ is updated at most once for each index pair (i,j). Thus the running time for the decision algorithm is $O(n^2 + nm \log n)$.

To extract a middle curve from the table X we additionally store labeled pointers in the table as follows. Each entry (i,j) in X that is set to $\texttt{true}$ (except for $(0,0)$) gets a pointer to a $\texttt{true}$ entry (i', j') with $i' < i$ and $j' < j$. Furthermore, the pointer receives two labels: $p|q$ and $f|b$ where the first indicates the curve P or Q, and the second indicates *forward* or *backward*. The two labels together exactly specify one of the four points $p_{i'}, q_{j'}, p_i, q_j$. A middle curve can be reconstructed from these pointers by following them from the entry (n,m) to $(0,0)$ and choosing the points according to the labels. The pointers can be set in the algorithm as follows: when $X[i,j]$ is set to $\texttt{true}$ in the $\neg X[i,j]$ case, a pointer is set to a true entry in $\hat{L}_P(i,j)$ or $\hat{L}_Q(i,j)$ labeled $(p|q, b)$. When $X[i', j']$ is set to true when adding an upper wedge $U_P(i,j)$ or $U_Q(i,j)$ in the $X[i,j]$ case, a pointer is set to (i,j) labeled $(p|q, f)$. Thus in both cases the pointer refers to p_i or q_j depending on whether the P or Q wedge is involved.

Note that the algorithm allows multiple occurrences of vertices. However, the restriction enforces that if a vertex occurs multiple times, then all vertices of the other curve that occur in between are matched to that vertex in the discrete Fréchet matching. Figure 5 shows an example of this.

4.2 Optimization Problems for Two Curves

The optimal distance is attained between a pair of points from $P \cup Q$. We therefore sort all distances between such pairs of points in $O(n^2 \log n)$ time and then find the smallest distance by binary search using the decision algorithm.

If we wish to minimize the size of an ordered middle curve for a fixed distance ε, we can, similar to the case in Section 3.2, adapt the dynamic program by storing the smallest size of a middle curve covering up to p_i and q_j in the arrays, and substituting $\vee$ by a min. In this case, however, we cannot use the upper envelope $\bar{X}$ anymore to facilitate fast queries and updates. So, in each iteration we instead update each entry in X individually, replacing the

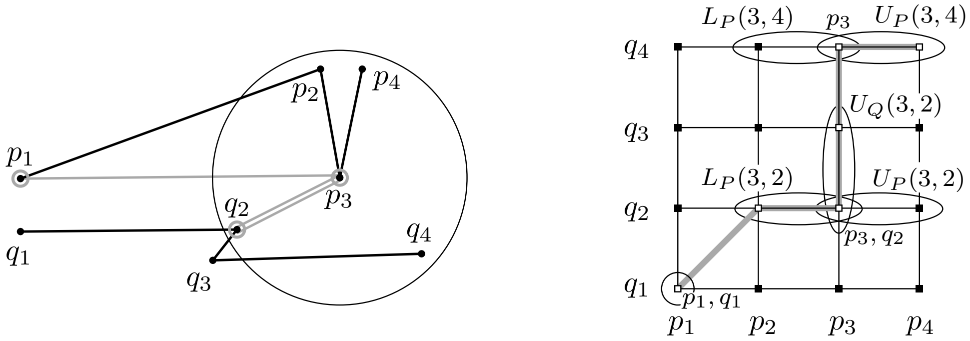


Figure 5: (a) Two curves $P = (p_1, p_2, p_3, p_4)$ and $Q = (q_1, q_2, q_3, q_4)$, and a restricted middle curve $R = (p_1, p_3, q_2, p_3)$ that uses p_3 twice. (b) A diagram illustrating array X and wedges.

"**if** $cl(p_i, q_j) = \texttt{true}$" statement with two nested for-loops. Hence, the total running time is $O((mn)^4)$. If we wish to optimize ε as well, we can do so using binary search, resulting in a total running time of $O((mn)^4 \log n)$.

4.3 Generalization to Multiple Curves

For $k > 2$ curves of size at most n each, the decision algorithm can be generalized to work with a $(k-1)$-dimensional range tree for $\bar{X}$ and running time $O(n^k \log^{k-1} n)$. We search over all distances between two points from any curves, so a middle curve can be decided in $O(n^k \log^k n)$ time. Using the pointers set by the algorithm, the algorithm can also output a middle curve.

To compute a minimal restricted middle curve for a fixed distance, we can modify the simpler $O(n^{2k})$ algorithm, where we iterate over all entries $X[i, j]$ and if reachable add the k upper wedges. We now store in each entry the minimum complexity needed to cover the curves up to here (and possibly a pointer to the last point added), and take the minimum when adding wedges. This simpler algorithm has complexity $O(n^{2k})$ and our technique of speeding up the algorithm using the upper envelope does not apply.

We summarize the results of this section in the following theorem:

Theorem 2. *For two polygonal curves with n and m vertices for $m \leq n$, and $\varepsilon > 0$, it can be decided whether there exists a restricted middle curve with distance at most ε in $O(n^2 + mn \log n)$ time. A restricted middle curve with minimum distance can be computed in $O(n^2 \log n + mn \log^2 n)$ time. A restricted middle curve with minimum complexity and distance at most $\varepsilon > 0$ to the input curves can be computed in $O((mn)^4)$ time, and ε can be optimized in total $O((mn)^4 \log n)$ time. For $k \geq 2$ curves of size at most n each, a restricted middle curve with minimal distance can be computed in $O(n^k \log^k n)$ time.*

5 Algorithms for the Unordered Case

In this section we consider the problem of computing an unordered middle curve. We present a straight-forward approach for the problem in Section 5.1. Then in Sections 5.2–5.4 we present faster algorithms for the case of curves in the Euclidean plane.

In the unordered case we can formulate the problem more generally, that we are given two curves P, Q and a set S of points from which we build the middle curve R. The algorithms we present in this section work for any set S of points. When we build R from P and Q we use $S = P \cup Q$.

5.1 Straight-Forward Decision and Optimization

Let again $P = (p_1, \ldots, p_n)$ and $Q = (q_1, \ldots, q_m)$ be two input curves with $m \le n$. Let S be a set of ℓ points (possibly $S = P \cup Q$) from which we build the middle curve R. To solve the decision problem for the unordered case, we modify the dynamic programming algorithm for computing the discrete Fréchet distance of two curves [6] as follows. Let $\varepsilon > 0$ be an input of the decision problem. We consider an $n \times m$ matrix X, which we call the *free space matrix*. Each entry $X[i,j]$ corresponds to the pair (p_i, q_j) of points. In contrast to the original algorithm, we mark an entry $X[i,j]$ `free` if and only if there exists a point v from S such that v has distance at most ε to both p_i and q_j. Then we search for a monotone path from $X[1,1]$ to $X[n,m]$ within the `free` entries in X.

One way to determine whether $X[i,j]$ is `free` for an index pair (i,j) is to test all possibilities for a point $v \in S$, each of which can be tested in $O(1)$ time. When $|S| = \ell$ the free space matrix of size $m \times n$ can be computed in $O(\ell mn)$ time. When $S = P \cup Q$, i.e., $\ell = m + n$ this results in $O(mn^2)$ time for $m \le n$. The running time for searching for a monotone path in the matrix is $O(mn)$.

Similarly, we can compute an unordered middle curve with minimal distance in the same time as follows. We let $X[i,j]$ be the minimum of $\max\{d(v,p_i), d(v,q_j)\}$ over all points $v \in S$. Then we search for a monotone path from $X[1,1]$ to $X[n,m]$ such that the maximum entry $X[i,j]$ in the path is minimized.

For $k \ge 2$ input curves of size at most n each, where k is constant, and $|S| = \ell$, we can compute an unordered middle curve of minimal distance in $O(\ell n^k)$ time in total using a k-dimensional free space matrix of size n^k.

To compute an unordered middle curve of minimal size for a given distance $\varepsilon > 0$, we consider the following graph: vertices are all tuples of points from the k curves, and there is a directed edge from $(i_1, \ldots, i_k)$ to $(j_1, \ldots, j_k)$ if there is a vertex in S that covers the subcurves from i_h to j_h on each curve $1 \le h \le k$. This graph has n^k vertices and up to n^{2k} edges. It can be constructed in $O(\ell n^{2k})$ time, by computing for each of n^k vertices all of its outgoing edges in $O(\ell n^k)$ time. In the graph we search for a shortest path from vertex $(1, \ldots, 1)$ to vertex $(n, \ldots, n)$ which corresponds to a minimum middle curve.

Theorem 3. *For $k \ge 2$ curves of size at most n each and a set S of ℓ points, an unordered middle curve built from S can be computed in $O(\ell n^k)$ time. For fixed $\varepsilon > 0$, an unordered middle curve with minimum complexity and distance at most ε to the input curves can be computed in $O(\ell n^{2k})$ time.*

5.2 Decision Algorithm for Two Curves in the Euclidean Plane

In this section, we assume that the two input curves lie in the Euclidean plane and use d_E to denote the Euclidean metric. A further assumption is that now $S = P \cup Q$. We describe how to determine whether $X[i,j]$ is `free` more efficiently for the decision problem in this setting.

We will use a circular sweep to determine, for each point q_j in Q, all points p_i in P such that $X[i,j]$ is `free`, i.e., there is some point v of $P \cup Q$ which has distance at most ε to both p_i and q_j. Let $U_j(\varepsilon)$ be the union of disks of radius ε centered at points in $P \cup Q$ and containing $q_j \in Q$, and $\partial U_j(\varepsilon)$ be the boundary of $U_j(\varepsilon)$. Then, for a point $p_i \in P$ contained in $U_j(\varepsilon)$, $X[i,j]$ is `free`. To compute $X[i,j]$ for all $p_i \in P$, we construct $U_j(\varepsilon)$ and perform a circular sweep around q_j for the points in P. Once the endpoints of the circular arcs of $\partial U_j(\varepsilon)$ and all points $p_i \in P$ are sorted around q_j in clockwise order, the circular sweep takes $O(m+n)$ time.

We design an algorithm that computes $U_j(\varepsilon)$ efficiently by constructing two data structures, the *history list* $\mathcal{H}_j$ and the *deletion list* $\mathcal{D}_j$. In the *preprocessing* phase, we increase ε gradually and consider the combinatorial structure of $U_j(\varepsilon)$. In doing so, we maintain the changes of the combinatorial structure of $U_j(\cdot)$ using the two data structures. In the *construction* phase, we compute, for a given ε, the union $U_j(\varepsilon)$ of disks using the two data structures. This will allow us to solve the decision problem efficiently. The construction phase takes $O(m+n) = O(n)$ time while the preprocessing phase takes $O(mn \log n)$ time. The space we use for the data structures is $O(mn)$.

Each arc of $\partial U_j(\varepsilon)$ changes continuously as ε increases. We will show that there are at most two arcs in $\partial U_j(\varepsilon)$, which come from the same circle (Lemma 6). We treat them as distinct elements in $\partial U_j(\varepsilon)$.

Data Structures for a Point $q_j \in Q$

1. The data structures we maintain during the preprocessing phase:
   (a) $S_j(\varepsilon)$ denotes the set of points of $P \cup Q$ that are within distance ε from the point q_j.
   (b) $\Gamma_j(\varepsilon)$ is the balanced binary search tree representing the union of disks of radius ε centered at points of $S_j(\varepsilon)$ for a fixed ε. The union $U_j(\varepsilon)$ of disks is star-shaped and its boundary $\partial U_j(\varepsilon)$ is a sequence of circular arcs with vertices in between. We maintain the balanced binary search tree $\Gamma_j(\varepsilon)$ of these circular arcs in clockwise order around q_j. Each element in $\Gamma_j(\varepsilon)$ corresponds to an element in the history list (defined below) and they reference each other with a pointer.
2. The data structures used for constructing $U_j(\varepsilon)$ in the construction phase:
   (a) The history list $\mathcal{H}_j = \{x_1, \ldots, x_l\}$: Each element of the list corresponds to an arc of $\partial U_j(\varepsilon)$ for some ε. Each such arc is part of a disk boundary of radius ε centered at a point of $S_j(\varepsilon)$ that appears on $\partial U_j(\varepsilon)$. This list represents the order of circular arcs of $\partial U_j(\varepsilon)$ for *every* $\varepsilon > 0$. That is, for any three elements in $\mathcal{H}_j$, if all arcs corresponding to the elements appear on $\partial U_j(\varepsilon)$ for some fixed $\varepsilon > 0$, then the order of them on $\partial U_j(\varepsilon)$ is the same as the order of the three elements in $\mathcal{H}_j$.
   (b) The deletion list $\mathcal{D}_j = \{(\varepsilon_1, \varepsilon_1'), \ldots, (\varepsilon_t, \varepsilon_t')\}$ with $\varepsilon_k \leq \varepsilon_k'$ for every $1 \leq k \leq t$: The k-th element of the list is defined by the k-th closest point in $P \cup Q$ from q_j. By Lemma 6, the disk centered at the k-th closest point of radius ε has at most two arcs appearing on $\partial U_j(\varepsilon)$ for any fixed $\varepsilon > 0$. An arc of the disk disappears from $\partial U_j(\varepsilon)$ at $\varepsilon = \varepsilon_k$, and the other arc disappears from $\partial U_j(\varepsilon)$ at $\varepsilon = \varepsilon_k'$. ($\varepsilon_k$ or ε_k' might be infinity or zero.) Since the list is an array of size $m + n$, we can access each element in $O(1)$ time.

Preprocessing Phase: Constructing the Data Structures. For each $q_j \in Q$, we construct the data structures mentioned above. To do this, we imagine that ε increases from zero to infinity. As ε changes, the combinatorial structure of $U_j(\varepsilon)$ changes. More specifically, a new arc appears on $U_j(\varepsilon)$ and an arc disappears from $U_j(\varepsilon)$. A value ε' is called an *event* if the combinatorial structure of $U_j(\varepsilon)$ changes at $\varepsilon = \varepsilon'$.

There are two types of events: *point events* and *radius events*. A new arc appears on $U_j(\varepsilon)$ only if a new point is inserted to $S_j(\varepsilon)$. A new point $p \in P \cup Q$ is inserted to $S_j(\varepsilon)$ only when ε is the distance between q_j and p by definition. We call such a value ε a *point event*. We say p *defines* this point event. For a point event e, we let $p(e)$ be the point defining e. There are two cases how an arc disappears from $U_j(\varepsilon)$: the case that the arc is contained in the disk centered at $p(e')$ with radius e' at $\varepsilon = \varepsilon'$ for a point event ε' and the case that $\partial U_j(\varepsilon)$ passes through the point equidistant from the centers of the arc and its two neighboring arcs of $U_j(\varepsilon')$ at some point

ε'. The first case is handled by the point event defined by p, and the second case is handled by a *radius event* which is defined as follows. For any three consecutive arcs of $\partial U_j(\cdot)$, we call $d(p, c)$ a *radius event*, where p is the center of any of the three arcs and c is the point equidistant from the centers of the three arcs. We say the three arcs *define* this event.

In the following, we show how to handle each event as ε increases.

1. Sort all points of P around q_j in clockwise order. Let $\mathcal{L}_j$ denote the sorted list.
2. Initialize $\mathcal{H}_j := \{q_j\}, \Gamma_j(0) := \{p_j\}, S_j(0) := \{p_j\}$ and $U_j(0) := \{p_j\}$. Initialize $\mathcal{D}_j$ to the array of size $m + n$ each of whose elements is initialized to a null value.
3. Sort all points of $P \cup Q$ in increasing order of distance from q_j and store the distances together with their corresponding points as events in an event queue $\mathcal{E}$. Note that they are point events.
4. While $\mathcal{E}$ is not empty, handle the earliest event $e \in \mathcal{E}$ as follows. Let e' denote the event we just handled.
   (a) If e is a point event, the boundary of the disk centered at $p(e)$ with radius e appears on $\partial U_j(e)$ in one connected circular arc γ if it appears on $\partial U_j(e)$ (see Lemma 4.) The endpoints of γ can be computed in $O(\log n)$ time by Lemma 5.

   Let γ_1 and γ_2 denote the neighboring arcs of γ, respectively, along $\partial U_j(e)$ so that γ_1, γ and γ_2 appear on $\partial U_j(e)$ in clockwise order. See Figure 6(a) for an illustration. When we compute γ, we can obtain γ_1 and γ_2. We find the element in $\mathcal{H}_j$ corresponding to γ_1 and insert an element corresponding to γ to $\mathcal{H}_j$ next to the element. We remove all arcs of $\partial U_j(e')$ coming from γ_1 to γ_2 in clockwise order along $\partial U_j(e')$ and update the corresponding elements in $\mathcal{D}_j$ to e. We update $\Gamma_j(e')$ accordingly to obtain $\Gamma_j(e)$ in $O(c \log n)$ time, where c is the number of the deleted arcs.

   Then we have new triples of consecutive arcs along $U_j(e)$, which induce radius events. Note that such a triple contains γ, and thus there are at most three new radius events. We insert all such events to $\mathcal{E}$.

   (b) If e is a radius event, we first check if all three arcs defining e appear on $U_j(e')$. If so, we remove the arc in the middle among the three arcs from $\Gamma_j(e')$ and update $\mathcal{D}_j$ accordingly by setting the value of the element corresponding to γ in $\mathcal{D}_j$ to e. Due to the deletion of γ, the two neighboring arcs of γ become adjacent in $U_j(e)$ for which we insert a new radius event.

The number of point events is $O(n)$ and the number of radius events is bounded by the number of distinct arcs appearing on $U_j(\varepsilon)$ over all increasing ε values, which is $O(n)$ by Lemma 6. Thus, the number of events in the preprocessing phase and the size of the data structures are $O(n)$. The preprocessing phase takes $O(n \log n)$ time for each $q_j \in Q$.

Construction Phase: Constructing the Free Space Matrix. Given $\varepsilon > 0$, the construction phase works as follows. For each $q_j \in Q$:

1. Scan the list $\mathcal{H}_j$ from the first element to the last element and check the list $\mathcal{D}_j$ to determine whether each arc appears on $\partial U_j(\varepsilon)$. In $O(n)$ time, we can obtain the sequence of the arcs appearing on $\partial U_j(\varepsilon)$ in clockwise order, which represents $\partial U_j(\varepsilon)$ itself.
2. Perform a circular sweep by a ray from q_j around q_j with the points in $\mathcal{L}_j$ and the vertices of $\partial U_j(\varepsilon)$. During the sweep, the ray always intersects an arc of $\partial U_j(\varepsilon)$. We can determine

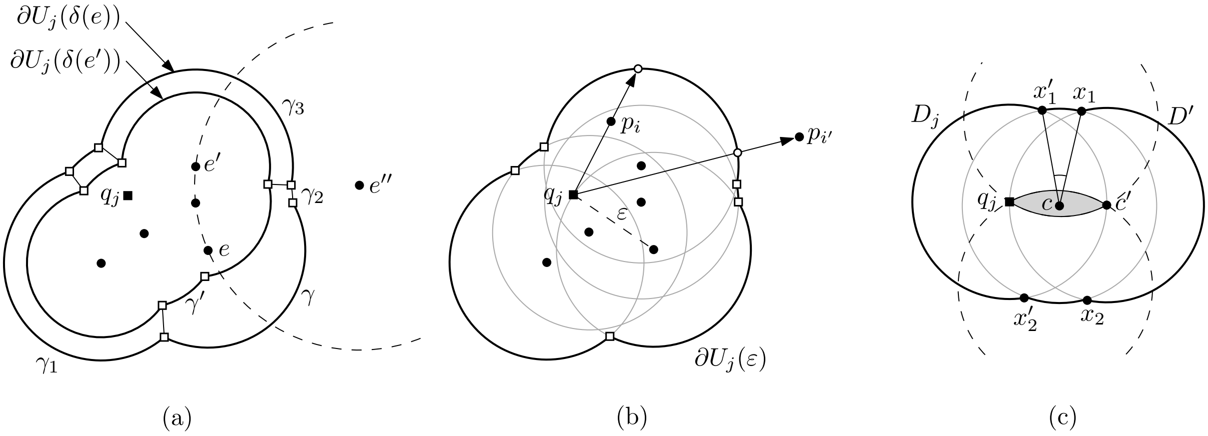


Figure 6: (a) Preprocessing. At the point event e, the arc γ' disappears and the new radius event e'' is created, where e'' is the point equidistant from the centers of γ_1, γ_2, and γ. γ_2 will disappear from the boundary at radius event e'' unless it disappears before the event. (b) Construction of the free space matrix. During the circular sweep, we compare each point $p_i \in P$ with the intersection point of $\partial U_j(\varepsilon)$ with the ray from q_j through p_i to determine whether p_i is in $U_j(\varepsilon)$. Here, $X[i, j]$ is `free` and $X[i, j']$ is not `free`. (c) The arc centered at c is subdivided into two subarcs by the event e. Then $\angle x_1 c x'_1$ and $\angle x_2 c x'_2$ are at most $2\pi/3$.

whether p_i is in $U_j(\varepsilon)$ by comparing each point p_i in $\mathcal{L}_j$ encountered by the ray with the current circular arc of $\partial U_j(\varepsilon)$ intersected by the ray. If so, set $X[i, j]$ to `free`. Figure 6(b) illustrates the circular sweep. This again can be done in $O(n)$ time once $\mathcal{L}_j$ has been computed in the preprocessing step.

For the correctness of the algorithm, we show the following lemma.

Lemma 4. *For a point event e, at most one arc of the disk of radius e centered at $p(e)$ appears on $\partial U_j(e)$.*

Proof. Assume to the contrary that there are two maximal circular arcs, γ and γ', on $\partial U_j(e)$ such that both arcs are on the boundary of the disk of radius e centered at $p(e)$. Let D_j be the disk of radius e centered at q_j. Since $d_E(p(e), q_j) = e$, the boundary of D_j contains $p(e)$. Then there must be a disk D' splitting the arc of $\partial D \setminus D_j$ into two, one containing γ and the other containing γ'.

Here, the center of D' is contained in D_j since the center of D' has already been handled. Thus $\partial D \cap D_j$ intersects D' at a point, say x_4. Since D' splits $\partial D \setminus D_j$ into two, there are three points x_1, x_2 and x_3 appearing on $\partial D \setminus D_j$ in clockwise order such that x_1 and x_3 are contained in D', and x_2 and x_4 are not contained in D'. This means that D and D' cross each other, which is a contradiction. ◻

To analyze the running time of the algorithm, we need the following lemma.

Lemma 5. *The endpoints of the circular arc to be inserted at step 4(a) of the preprocessing phase can be computed in $O(\log n)$ time.*

Proof. Let e and e' be the current event and the event previous to e, respectively. We maintain $\Gamma_j(\varepsilon)$, which is the balanced binary search tree of the arcs of $\partial U_j(\varepsilon)$ in clockwise order around q_j. Note that $U_j(\varepsilon)$ is star-shaped with respect to q_j and there is no structural change to $\partial U_j(\varepsilon)$ for $e' \leq \varepsilon < e$. Since the disk of radius e centered at $p(e)$ is also star-shaped and contributes only one connected circular arc γ to $\partial U_j(e)$, the two endpoints of γ can be computed in $O(\log n)$ time by a binary search on the vertices of $\partial U_j(e')$. ◻

Lemma 6. *Once an arc is divided into two subarcs, the subarcs will never be divided again.*

Proof. Let γ_1 and γ_2 be the arcs appearing on $\partial U_j(\varepsilon)$ for some $\varepsilon > 0$, which come from the same disk D centered at a point, say c. Let x_1, x_1', x_2 and x_2' be the endpoints of γ_1 and γ_2 in clockwise order along the disk centered at c with radius ε.

We first claim that $\angle x_1cx_1'$ and $\angle x_2cx_2'$ are at most $2\pi/3$. See Figure 6(c). Let D_j be the disk centered at q_j with radius ε. Since there are two arcs which come from ∂D, there is a disk, say D', splitting $\partial D \setminus D_j$. Note that the center of D', say c', is contained in D_j. Without loss of generality, assume that q_jc' is contained in the x-axis. Since D' splits $\partial D \setminus D_j$, D contains the intersection points between $\partial D'$ and ∂D_j. Thus the center c of D is contained in the intersection I of D', D_j and the disks centered at the intersection points in $\partial D' \cap \partial D_j$ of radius ε. Note that I consists of two circular arcs whose common endpoints are q_j and c'. Also, notice that each of x_1, x_1', x_2 and x_2' lies on the bisector of c and q_j (or c'). By construction, the intersection point between the two bisectors, one between c and q_j and one between c and c', lies outside of $D \cup D_j$. Therefore, $\angle x_1cx_1'$ and $\angle x_2cx_2'$ are at most $2\pi/3$.

Now we show that γ_1 is not divided further. The case of γ_2 can be shown analogously. Assume to the contrary that γ_1 is divided at ε. Again, let D_j be the disk centered at q_j with radius ε. This means that there is a disk, say D'', centered at a point in D_j, say c'', with radius ε such that x_1 and x_1' are not contained in D'' but a point, say x, other than its endpoints is contained in D''. Imagine the set of points whose distance to x_1 (and x_1') is larger than ε and whose distance to x is at most ε. The set lies outside of D because $\angle x_1cx_1'$ is at most $2\pi/3$. Since c'' lies in D_j and lies outside of D, the line segment cc'' intersects $\partial D \setminus \gamma_1$. This means that there are four points on ∂D such that the first and third points are contained in D'' but the second and fourth points are not contained in D'' in clockwise order around ∂D. This contradicts that D and D'' are disks. Therefore, γ_1 is not divided further, and the lemma holds. ◻

To obtain a covering sequence in addition to a yes-answer, for each entry $X[i, j]$ of the free space matrix, we mark the center of the circular arc of $\partial U_j(\varepsilon)$ intersected by the ray starting from q_j towards p_i. Then the sequence of labels of a monotone path gives a feasible unordered sequence for the middle curve.

Theorem 7. *For two polygonal curves with n and m vertices for $m \leq n$ in the Euclidean plane, the decision problem for the unordered case can be solved in $O(mn)$ time with $O(mn \log n)$ preprocessing time. A covering sequence can be computed in the same time.*

5.3 Optimization Algorithm for Two Curves in the Euclidean Plane

We apply binary search on the sorted list of distances of pairs of points from $P \cup Q$ involved in each step. There are $O((m+n)^2) = O(n^2)$ distinct distances each defined by two points from $P \cup Q$. We will show that we need only $O(mn)$ of them to compute the optimal distance ε^*. The optimization algorithm we propose works as shown in the following four steps.

1. Compute the set $\mathcal{D}$ of distances each defined by two points that are either both from Q, or one from P and one from Q.

2. Sort the $O(mn)$ distances of $\mathcal{D}$ and apply binary search on the sorted list with the decision algorithm in Section 5.2. Let ε_1 be the largest distance of $\mathcal{D}$ that the decision algorithm

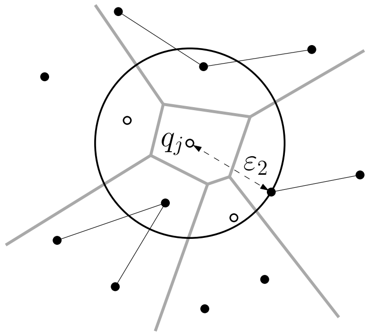


Figure 7: The white (circle) points are in Q and the black points are in P. The lengths of the line segments connecting two black points are candidates of ε^*.

returns "no" and ε_2 be the smallest distance of $\mathcal{D}$ that the decision algorithm returns "yes". Then we know that $\varepsilon_1 \leq \varepsilon^* \leq \varepsilon_2$. If $\varepsilon_1 \neq \varepsilon^*$ and $\varepsilon_2 \neq \varepsilon^*$, then ε^* is the distance defined by two points in P. See Figure 7.

3. To find ε^*, for each point $q_j \in Q$,

   (a) compute the set S_j of points in $P \cup Q$ that are at distance at most ε_2 from q_j, and construct the Voronoi diagram $\mathrm{VD}(S_j)$.

   (b) For each point p_i in $P \setminus S_j$, locate the cell of $\mathrm{VD}(S_j)$ that contains p_i. If the site x associated with the cell is from P and $\varepsilon_1 < d_E(p_i, x) < \varepsilon_2$, then $d_E(p_i, x)$ is a candidate for ε^*.

4. Sort the $O(mn)$ candidate distances and again apply binary search on the sorted list with the decision algorithm above.

Analysis. Let (p_i, q_j, x) be a tuple realizing ε^*. Then $\max\{d_E(p_i, x), d_E(q_j, x)\} = \varepsilon^*$. Clearly, x is the point in $P \cup Q$ that minimizes $\max\{d_E(p_i, x), d_E(q_j, x)\}$. If $x \in P$ and $\varepsilon_1 < \varepsilon^* < \varepsilon_2$, then $d_E(p_i, x) > d_E(q_j, x)$. Thus x is the point in S_j that is closest to p_i. Thus, x is the point site associated with the Voronoi cell in $\mathrm{VD}(S_j)$ that contains p_i. This proves that ε^* is in the set of all candidates.

Let us analyze the running time of the optimization algorithm. The set $\mathcal{D}$ can be constructed in $O(mn)$ time. It takes $O(mn \log n)$ time to sort the distances in $\mathcal{D}$. The binary search on the sorted list with the decision algorithm takes $O(mn \log n)$ time as the preprocessing phase is executed only once for each $q_j \in Q$ and the history and deletion lists are used for different radii. In Step 3, the Voronoi diagram $VD(S_j)$ can be constructed in $O(n \log n)$ time for each $q_j \in Q$, and the point location for n points can be performed in the same time. Step 3(b) takes $O(n \log n)$ time for each $q_j \in Q$.

5.4 Generalization to Multiple Curves in the Euclidean Plane

The decision algorithm can be extended to k curves $P^1, \ldots, P^k$ of size at most n each for a constant k in the Euclidean plane. We construct a k-dimensional free space matrix whose entries correspond to k-tuples of points from distinct curves. An entry of the matrix is marked as `free` if there is an input point in the intersection of the disks centered at points in the k-tuple corresponding to the entry with radius ε, where ε is an input distance for the decision problem.

To construct the matrix, we use an approach similar to the one for $k = 2$. For every $(k-1)$-tuple $(p_1, \ldots, p_{k-1})$ with $p_i \in P^i$ for $i = 1, \ldots, k-1$, we do the following. Let D be the intersection of the disks centered at p_i with radius ε for all $i = 1, \ldots, k-1$. We compute the union U of the disks with radius ε centered at input points lying in D, and check for each point in P^k whether it is contained in the union. Here, the boundary of U has the star-shaped property for any point p_i in the $(k-1)$-tuple. We mark the entry in the matrix corresponding to the k-tuple $(p_1, \ldots, p_k)$ as `free` if and only if p_k is contained in the union. Then we check if there is a monotone path from $X[1, \ldots, 1]$ to $X[n, \ldots, n]$ within the `free` entries in X.

The construction of U takes $O(kn \log(kn))$ time for fixed $\varepsilon > 0$. However we can compute it more efficiently by maintaining the history data structure as we did for $k = 2$. Imagine that ε increases from zero to infinity, and consider the combinatorial changes of U. There are two types of events: the point events and the radius events. The radius event is defined in the same way as the case of $k = 2$. For the point events, observe that the centers of the circular arcs of ∂U are in D. As ε increases, D changes as well. A new arc appears on ∂U when its center appears on the boundary of D. Also, its center appears on the boundary of D when ε is the distance between the center and the point in the $(k-1)$-tuple farthest from the center. Such distances are defined as point events. Clearly, there are $O(n)$ point events. Also, Lemmas 4, 5 and 6 hold for a larger k. Thus we can maintain the combinatorial structure of U in $O(kn \log(kn)) = O(n \log n)$ time. We keep track of the combinatorial changes using the history and deletion data structures as we did for $k = 2$. Then after the preprocessing, we can construct the union in $O(kn) = O(n)$ time. We can check for each point in P^k whether it is contained in the union in $O(n)$ time. Thus, the decision algorithm takes $O(n^k)$ time once the history data structures are constructed for all $(k-1)$-tuples.

To compute a middle curve, we first construct history data structures for all $(k-1)$-tuples in $O(n^k \log n)$ time. Then we sort all distances defined by point pairs from $P^1 \cup \ldots \cup P^k$ and search the optimal distance among them. Thus, we can compute an optimal covering sequence in $O(n^k \log n)$ time.

Theorem 8. *For two polygonal curves with n and m vertices for $m \leq n$ in the Euclidean plane, the optimization problem for the unordered case can be solved in $O(mn \log n)$ time. An optimal covering sequence can be computed in the same time. For a fixed $k \geq 2$, the optimization of k curves of size at most n each in the Euclidean plane can be solved in $O(n^k \log n)$ time.*

6 Discussion

We presented algorithms for computing a middle curve of minimal discrete Fréchet distance to the input curves. All our algorithms run in time exponential in k, the number of input curves. Hence these are practical only for small k. However, other algorithms that compute variants of the Fréchet distance for k curves such as [5] and [7] also take time exponential in k due to the use of a k-dimensional free space diagram. Assuming the Strong Exponential Time Hypothesis it is known that essentially no faster algorithms are possible [3]. Hence we also do not expect any substantially faster algorithms for finding a middle curve based on the (discrete) Fréchet distance. An interesting open problem is to find more efficient approximation algorithms.

Also note that a middle curve (unordered, ordered, or restricted) computed by our algorithms can have complexity at most $nk - k + 1$. This follows because each vertex of the middle curve "advances" by at least one vertex on one of the input curves, of nk vertices in total. More formally, let R be a middle curve of size r. Consider the discrete Fréchet mappings of R to each of the input curves. Each of these gives a segmentation with duplicates of size r of the input curve. Now consider the set of all segmentations. For two consecutive vertices of R, the

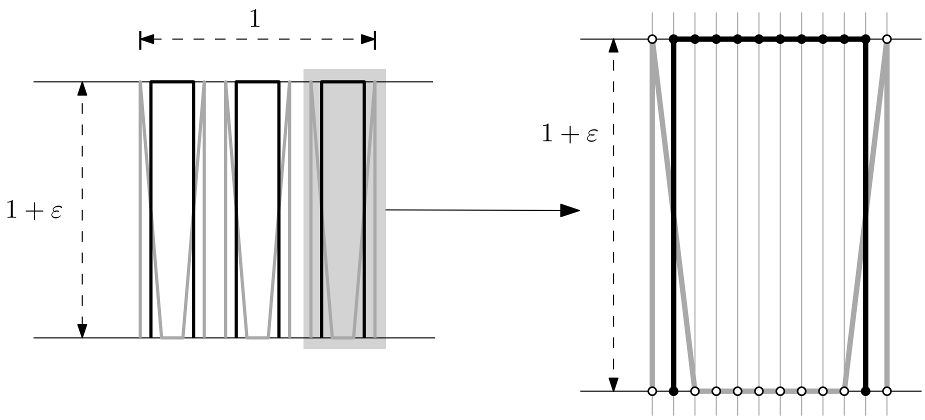


Figure 8: Example of a middle curve of complexity $nk/2$ (here $k = 6$). It consists of $k/2$ (here 3) gray curves and $k/2$ black curves. The curves are given in pairs of one gray and one black curve as shown on the right. Each gray curve has $n - 2$ vertices on the bottom line and 2 vertices on the top line whereas each black curve has 2 vertices on the bottom line and $n - 2$ vertices on the top line. The vertices are aligned along the two parallel lines.

corresponding sets of segmentations differ, i.e., one of the input curves advances by at least one vertex. Furthermore, the first vertex of the middle curve covers all first vertices of each of the input curves. Hence these do not increase the size of the output middle curve.

We observe that this bound is essentially tight by giving an example of k curves of complexity n each where a minimal middle curve has complexity $nk/2$. Consider the curves shown in Figure 8. In this example the only middle curve achieving the minimal Fréchet distance $1 + \varepsilon$ is the set of all points on the bottom line, either ordered from left to right or from right to left. For this, first observe that for a pair of corresponding gray and black curve, a middle curve consists of n points, one of each pair of points at distance $1 + \varepsilon$, arbitrarily from the top or bottom line. However a middle curve for all k curves may only contain points on the bottom line, which are (horizontally) close to the start and end point of all other curves.

However, in practice we expect middle curves to be much smaller. In the example in Figure 8 we observe that if we increase the distance, the complexity decreases fairly quickly. On the other hand, if we decrease the distance, then a middle curve is no longer possible.

Acknowledgments. This work was initiated at the 17^{th} 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] H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *International Journal of Computational Geometry & Applications*, 5(1-2):75–91, 1995.

[2] K. Buchin, M. Buchin, J. Gudmundsson, M. Löffler, and J. Luo. Detecting commuting patterns by clustering subtrajectories. *International Journal of Computational Geometry & Applications*, 21(3):253–282, 2011.

[3] K. Buchin, M. Buchin, W. Mulzer, M. Konzack, and A. Schulz. Fine-grained analysis of problems on curves. In G. Barequet and E. Papadopoulou, editors, *Proceedings of the 32nd European Workshop on Computational Geometry (EuroCG'16)*, 2016.

[4] K. Buchin, M. Buchin, M. van Kreveld, M. Löffler, R. I. Silveira, C. Wenk, and L. Wiratma. Median trajectories. *Algorithmica*, 66(3):595–614, 2013.

[5] A. Dumitrescu and G. Rote. On the Fréchet distance of a set of curves. In *Proceedings of the 16th Canadian Conference on Computational Geometry (CCCG'04)*, pages 162–165, 2004.

[6] T. Eiter and H. Mannila. Computing discrete Fréchet distance. Technical report, Technische Universität Wien, 1994.

[7] S. Har-Peled and B. Raichel. The Fréchet distance revisited and extended. *ACM Transactions on Algorithms*, 10(1):3:1–3:22, Jan. 2014.

[8] E. Sriraghavendra, K. Karthik, and C. Bhattacharyya. Fréchet distance based approach for searching online handwritten documents. In *Proceedings of the Ninth International Conference on Document Analysis and Recognition (ICDAR'07)*, volume 1, pages 461–465. IEEE Computer Society, 2007.

[9] M. J. van Kreveld, M. Löffler, and F. Staals. Central trajectories. In *31st European Workshop on Computational Geometry (EuroCG'15)*, pages 129–132, 2015.

[10] H. Zhu, J. Luo, H. Yin, X. Zhou, J. Z. Huang, and F. B. Zhan. Mining trajectory corridors using fréchet distance and meshing grids. In *Advances in Knowledge Discovery and Data Mining*, pages 228–237. Springer Berlin Heidelberg, 2010.