A Middle Curve Based on Discrete Fréchet Distance

Hee-Kap Ahn[1], Helmut Alt[2], Maike Buchin[3], Eunjin Oh[1(✉)], Ludmila Scharf[2], and Carola Wenk[4]

[1] Pohang University of Science and Technology, Pohang, Korea
{heekap,jin9082}@postech.ac.kr
[2] Free University of Berlin, Berlin, Germany
{alt,scharf}@mi.fu-berlin.de
[3] Ruhr University Bochum, Bochum, Germany
maike.buchin@rub.de
[4] Tulane University, New Orleans, USA
cwenk@tulane.edu

Abstract. Given a set of polygonal curves we seek to find a middle curve that represents the set of curves. We require that the middle curve consists of points of the input curves and that it minimizes the discrete Fréchet distance to the input curves. We present algorithms for three different variants of this problem: computing an ordered middle curve, computing an ordered and restricted middle curve, and computing an unordered middle curve.

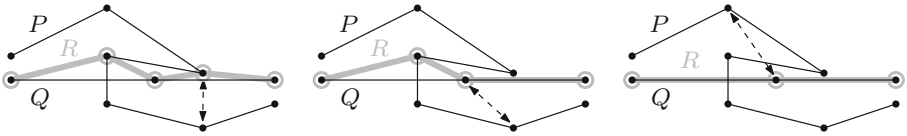
1 Introduction

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 provides a concise representation of the data, which is useful for data analysis and for reducing the size of the data, possibly by many magnitudes.

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 path of the individuals, hence we ask for it to minimize the discrete Fréchet distance to these. The Fréchet distance [1] and the discrete Fréchet distance [5] are well-known distance measures, which have been successfully used before in analyzing handwritten characters [7] and trajectories [2,9].

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. Work by Ahn and Oh was supported by the NRF grant 2011-0030044 (SRC-GAIA) funded by the government of Korea.

E. Kranakis et al. (Eds.): LATIN 2016, LNCS 9644, pp. 14–26, 2016.
DOI: 10.1007/978-3-662-49529-2_2



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

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

We consider three variants of this problem, which we introduce now more formally for two curves. Given two point sequences P and Q, we wish to compute a *middle curve* R consisting of points from $P \cup Q$ that minimizes $\max(d_F(R, P), d_F(R, Q))$, where d_F denotes the discrete Fréchet distance. In the following 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 a 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. And we call an ordered middle curve R *restricted*, if points on R are mapped to themselves in a matching realizing the discrete Fréchet distance. Recall that points from R originate from P or Q, hence this seems a natural restriction. Furthermore, we distinguish whether points may occur *multiple times* or not (but still respecting the order/restriction if applicable).

Figure 1 illustrates the three cases we consider: the ordered, restricted, and unordered cases. Note how adding the restrictions (from unordered to restricted) changes the middle curve and increases the distance to the input curves. Requiring to respect the order of the input curves seems very natural. However, as we will see, the unordered case allows for the most efficient algorithm. Matching a vertex to itself on the middle curve is also natural. Furthermore, we will see that the restricted case allows for a more efficient algorithm.

Related Work. Several papers [3,6,8] 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] and Kreveld et al. [8] both require the middle curve to use parts of edges of the input. Buchin et al. aim 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. Kreveld et al. aim to be as close as possible to all trajectories at any time and allow 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 [6] 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".

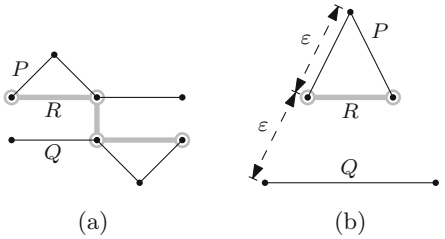


Fig. 2. (a) The middle curve may need to consist of vertices from both curves. (b) The 2-approximation is tight.

2-*Approximation.* A simple observation is that any of the input curves is a 2-approximation for minimizing the distance, which follows by triangle inequality. The 2-approximation is tight, as the example in Fig. 2 shows. We observe, however, that for an optimal middle curve we may need to choose a subset of vertices from both curves.

Our Results. We present algorithms for three variants of this problem for $k \geq 2$ curves of size at most n each:

1. **Ordered case:** An $O(n^{2k})$-time algorithm for computing an optimal ordered middle curve.
2. **Restricted case:** An $O(n^k \log^k n)$-time algorithm for computing an optimal restricted middle curve.
3. **Unordered case:** An $O(n^k \log n)$-time algorithm for computing an optimal unordered middle curve.

In the following sections, we present the algorithms for these cases. The algorithms for the restricted and the unordered cases allow points to appear multiple times. In the ordered case, we give algorithms for both.

Note that all algorithms run in time exponential in k, the number of trajectories. Hence these are practical only for small k. Other algorithms that compute variants of the Fréchet distance for k curves such as [4] and [6] also take time exponential in k due to the use of a k-dimensional free space diagram. Hence we do not expect faster algorithms for finding a middle curve based on the (discrete) Fréchet distance.

2 Algorithm for the Ordered Case

Here we present a dynamic programming algorithm for computing an ordered middle curve R. We first consider the case of two input curves $P = (p_1, \ldots, p_n)$ and $Q = (q_1, \ldots, q_m)$, and we do not allow multiple occurrences of the same point on R. Later we show how to generalize the algorithm to multiple input curves and to allow multiple point occurrences. Let $P_i, 1 \leq i \leq n$ denote the *prefix* $(p_1, ..., p_i)$ of P, where P_0 is defined as the empty sequence.

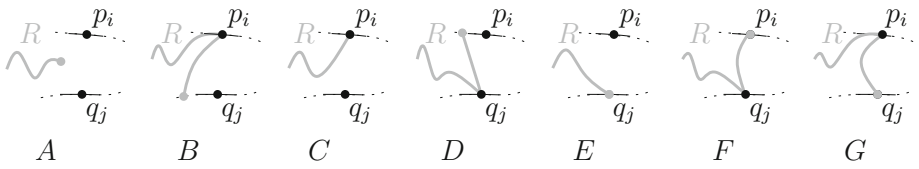


Fig. 3. Illustration of cases in the dynamic programming.

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

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

We say in this case that R *covers* P_k and Q_l (at distance ε). 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, we can represent the array X as the component-wise disjunction of seven Boolean arrays

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

For each array defined below, 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. See Fig. 3 for an illustration of the seven different cases that can occur. Our dynamic programming algorithm is based on these recursive identities for $i,j,k,l \ge 0$:

$$\begin{aligned}
A[0,0,0,0] &= \texttt{true} \\
A[0,0,k,l] &= \texttt{false} \text{ for } k \ge 1 \text{ or } l \ge 1 \\
A[i,0,k,l] &= X[i-1,0,k,l] \\
A[0,j,k,l] &= X[0,j-1,k,l] \\
A[i,j,k,l] &= X[i-1,j-1,k,l] \\
B[i,0,k,l] &= B[0,j,k,l] = \texttt{false} \\
B[i,j,k,l] &= G[i,j-1,k,l] \vee B[i,j-1,k,l]
\end{aligned}$$

The first equality for B holds 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 $cl(p, q)$ denote the truth value of $\|p - q\| \leq \varepsilon$, for two points p and q. The following identities hold for C.

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

The first two equalities hold because only an empty middle curve can cover 0 points. The equality for $C[i, j, k, l]$ models the two cases of whether the final point p_i in R covers p_k and q_l only, or whether it also covers 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 of F have similar explanations as the ones of C:

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

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 runs in time $O(n^2m^2)$, which is the size of each of the seven arrays. 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-tuple(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.

For the optimization problem, we can adapt a dynamic programming to compute the minimal value such that a covering middle curve exists. For this, X takes the minimum value of A to G; initialization is to $0|\infty$ instead of `true|false`; $\vee$ becomes min, and $\wedge$ becomes max. In this way we can solve the optimization problem in the same time as the decision problem.

The decision and optimization algorithms can be generalized to k sequences $P^1, ..., P^k$. The running time in this case is $O(n_1^2 ... n_k^2)$ for constant k, but the number of arrays is $2^{k-1}k + 2^k - 1$. The dynamic programming algorithm can also be modified to allow multiple occurrences of points on R, which requires distinguishing slightly more cases than before. Note that the length of a middle curve is at most nk if points may not appear multiple times, and at most $2nk$ if they may appear multiple times. The latter bound follows from a longest monotone path in the array of size n^{2k}.

Theorem 1. *For two polygonal curves with m and n vertices, the optimization problem for the ordered case can be solved in $O(m^2n^2)$ time. An optimal covering sequence can be computed in the same time. For $k \geq 2$ curves of size at most n each, the optimization can be solved in $O(n^{2k})$ time.*

3 Algorithm for the Restricted Case

Now we consider the case where the reparameterizations 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.

For this, we define arrays similar to Sect. 2. 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`. Similar to Sect. 2 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 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 or may not contain q_j in its interior).
$E'[i,j]$: R ends in q_j (and may or may not contain p_i in its interior).

In contrast to Sect. 2, 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).

We compute all $X[i,j]$ incrementally for increasing j and increasing i using dynamic programming. Consider p_i being matched to q_j. We use the *upper* wedge $U_P(i,j)$ to describe the resulting coverage of P and Q. Specifically, $U_P(i,j)$ denotes the set of index pairs (i',j') such that $\|p_{i''} - p_i\| \leq \varepsilon$ and $\|q_{j''} - p_i\| \leq \varepsilon$ for all $i \leq i'' \leq i'$ and $j \leq j'' \leq j'$. That is, $U_P(i,j)$ consists of the connected set of index pairs $(i',j') \geq (i,j)$ that are covered by p_i. The *lower* wedge $L_P(i,j)$ denotes the set of index pairs (i',j') such that $\|p_{i''} - p_i\| \leq \varepsilon$ and $\|q_{j''} - p_i\| \leq \varepsilon$ for all $i' \leq i'' \leq i$ and $j' \leq j'' \leq j$. Furthermore, we define the *extended lower wedge* $\hat{L}_P(i,j)$ which, in addition to all points 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 point pairs $(p_{i'}, q_{j'})$ for which $p_{i'}$ and $q_{j'}$

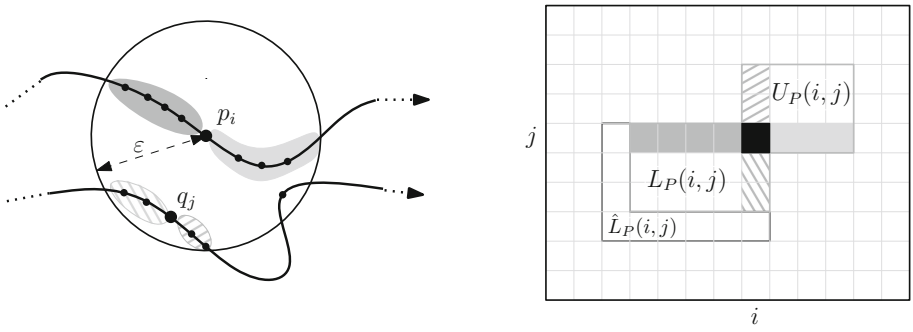


Fig. 4. Illustration of the wedges.

are both close to q_i. Figure 4 illustrates these wedges for a pair (i, j). Using this terminology we observe:

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

During the dynamic programming, in order to efficiently compute the values $X[i,j] = A'[i,j] \vee C'[i,j] \vee E'[i,j]$ we maintain the upper envelope $\bar{X}$ of all true elements in X. More specifically, we define $\bar{X}[i] = \max\{j \mid X[i,j] = \texttt{true}\}$. Note that X as well as $\bar{X}$ change during the dynamic programming for increasing j and i.

We store $\bar{X}$ in an augmented balanced binary search tree sorted on i. Each leaf corresponds to an index i and stores $\bar{X}[i]$. Each internal node v represents the interval of indices stored in the leaves of the subtree rooted at v, and stores two key values $m[v]$ and $M[v]$. Here, $m[v]$ is the minimum of all $\bar{X}[i]$ over all leaves i in the subtree rooted at v, and $M[v]$ is the maximum.

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 pair (i, j) such that $j_B \leq \bar{X}[i]$ and $i_B \leq i \leq i_T$. 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 the node i_T, we check the maximum key value of the left child v_L of v. The interval corresponding to v_L is fully contained in the interval $[i_B, i_T]$. Thus, if $M[v_L] \geq j_B$, the correct answer for the query is "yes". Otherwise, we do not need to consider the subtree rooted v_L further. Whenever we descend into the left child at a node on the path to i_B, we check the answer for the query analogously. Hence we can answer the query while we traverse the two paths, which takes logarithmic time.

2. *Updating* $\bar{X}$. We are given an upper wedge whose bottom-left corner is (i_B, j_B) and top-right corner is (i_T, j_T). We need to update $\bar{X}[i]$ to j_T for all $i_B \leq i \leq i_T$, if $\bar{X}[i] < j_T$.

   We traverse the balanced binary search tree from the root as follows. Assume that we reach a node v. If $j_T \leq m[v]$ or the interval corresponding to v does not intersect $[i_B, i_T]$, then we do not need to update the values of the leaf nodes in the subtree rooted at v. Hence we do not traverse this subtree. If $m[v] < j_T$ and the interval corresponding to v intersects $[i_B, i_T]$, 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 nodes, which will be updated. Then we go back to the root from those leaf nodes 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.

The algorithm consists of two parts: constructing all wedges and constructing the free space matrix X.

Constructing All Wedges. We construct the wedge $U_P(i, j)$ as follows: For fixed p_i, we first find the largest $k \geq i$ such that all $p_i, \ldots, p_k$ are in the disk of radius ε around p_i. Then we find the largest $l \geq j$ such that all $q_j, \ldots, q_l$ are in the disk of radius ε around 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 $j \leq j' \leq l$. Hence, all wedges $U_P(i, j)$ can be computed in $O(m + n)$ 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.

Constructing the Free Space Matrix X. First, initialize all $X[i, j]$ to `false`, except for $X[0, 0]$ which is set to `true`. Then compute $X[i, j]$ for $j = 1$ to m and for $i = 1$ to n. In each iteration, we process (p_i, q_j) only if they can be matched to each other, i.e., if $cl(p_i, q_j)$.

If $X[i, j]$ is `false`, i.e., we do not yet know of a middle curve covering P_i and Q_j, we first check whether adding p_i or q_j to a covering sequence extends 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`. Since in this case q_j can be added in addition, we set a flag in $X[i, j]$ to P, indicating that p_i has to be added first. 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, setting a flag for q_j this time.

If $X[i, j]$ is `true`, then both p_i or q_j can be added to a covering sequence, 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

$\max(j_B, \bar{X}[i']) \leq j' \leq j_T$ 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 touched only once, and at that time it is set to true.

The algorithm can now be summarized as follows.

Set $X[i,j] =$ `false` for all index pairs (i,j) except $X[0,0]$ which is set to `true`.
for $j = 1$ **to** m **do**
  for $i = 1$ **to** n **do**
    if $cl(p_i, q_j)$**:**
      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 `true and set the according flag in` $X[i,j]$.
      if $X[i,j]$**:** `Add` $U_P(i,j)$ `and` $U_Q(i,j)$ `to` X `and` $\bar{X}$.

Analysis. 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.

The running time for computing all wedges is $O((m+n)^2)$ 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 may perform a query on $\bar{X}$ which takes $O(\log(mn))$ time, and we may 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. 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 $O((m+n)^2 + mn\log(mn))$.

Lemma 1. *For two polygonal curves with m and n vertices, the decision problem for the restricted case can be solved in $O((m+n)^2 + mn\log(mn))$ time.*

Note that the algorithm allows multiple occurrences of vertices. However it restricts 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.

Optimization. The optimal distance will take one of the distances between pairs of points from $P \cup Q$, hence we first sort all distances in $O((m+n)^2 \log(mn))$ time and again search over them using the decision algorithm.

Lemma 2. *An optimal covering sequence for the restricted case can be computed in $O((m+n)^2 \log(mn) + mn\log^2(mn))$ time.*

Several Curves. For $k > 2$ curves the decision algorithm works the same with a $k-1$ dimensional range tree for $\bar{X}$ and runtime $O(n^k \log^{k-1} n)$. We again search over all distances between two points from any curves, so the optimal middle curve can be computed in $O(n^k \log^k n)$ time.

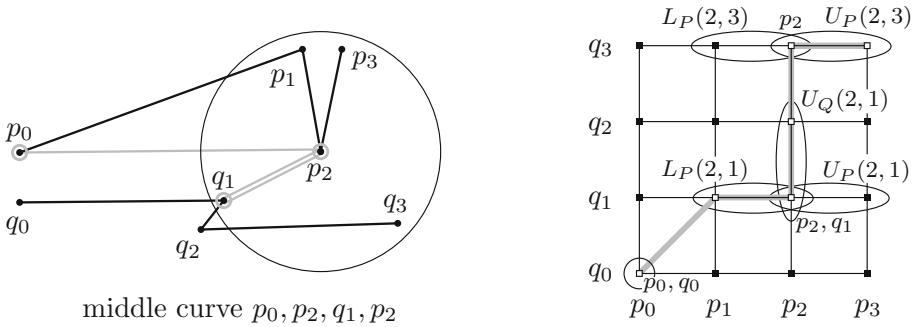


Fig. 5. An example of a middle curve R that uses a vertex (p_2) multiple times.

Output a Middle Curve. Using the pointers set by the algorithm, the algorithm can also output a middle curve. Note that a middle curve computed by the algorithm may have up to $2nk$ vertices. This follows from the algorithm because at each (i, j) at most two vertices (p_i and q_j) are added, and the length of a longest monotone path in the n^k grid is nk.

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

4 Algorithm for the Unordered Case

Let again $P = (p_1, \ldots, p_n)$ and $Q = (q_1, \ldots, q_m)$ be two input curves. To solve the decision problem for the unordered case, we modify the dynamic programming algorithm for computing the discrete Fréchet distance of two curves [5] as follows. We consider the $n \times m$ 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 color an entry $X[i, j]$ *free* if and only if there exists a point v from P or Q such that v has distance at most ε to both p_i and q_j. Then we search for a monotone path within the free entries in X.

4.1 Algorithm for the Decision Problem

We describe how to compute the labels more efficiently for the decision problem. Here, we use a circular sweep to determine for each point p_i all points q_j such that $X[i, j]$ is free, i.e., there is some point v of P or Q which has distance at most ε to both p_i and q_j. Let $U_{p_i}(\varepsilon)$ be the union of disks of radius ε centered at points in $P \cup Q$ and containing p_i. Then, for a point $q_j \in Q$ contained in $U_{p_i}(\varepsilon)$, $X[i, j]$ is free. To compute $X[i, j]$ for all $q_j \in Q$, we construct $U_{p_i}(\varepsilon)$ and

perform a circular sweep around p_i for all points in Q. Once the circular arcs of the boundary $\partial U_{p_i}(\varepsilon)$ and all points $q_j \in Q$ are sorted along p_i in clockwise fashion, the circular sweep takes $O(m+n)$ time.

We design an algorithm that computes $U_{p_i}(\varepsilon)$ efficiently by constructing two data structures, called the *history* $\mathcal{H}_{p_i}$ and the *deletion list* $\mathcal{D}_{p_i}$. In the *preprocessing* phase, we gradually increase ε and construct the two data structures. When a fixed ε is given, we compute $U_{p_i}(\varepsilon)$ using the two data structures in the *construction* phase. This will allow us to solve the optimization problem efficiently in Sect. 4.2. The construction phase takes $O(m+n)$ time while the preprocessing phase takes $O(mn\log(mn))$ time. The space we use for the data structures is $O(mn)$.

In this extended abstract we give a sketch of the algorithm. The details will be presented in a full version of this paper.

The data structures for a point $p \in P$.

1. The history list $\mathcal{H}_{p_i} = \{x_1, \ldots, x_l\}$: This list represents the order of circular arcs of $\partial U_{p_i}(\varepsilon)$ for all $\varepsilon > 0$. For any three elements in $\mathcal{H}_{p_i}$, if all arcs corresponding to the elements appear on $\partial U_{p_i}(\varepsilon)$ for some $\varepsilon > 0$, then the order of them on $\partial U_{p_i}(\varepsilon)$ is the same as the one in $\mathcal{H}_{p_i}$.
2. The deletion list $\mathcal{D}_{p_i} = \{(\varepsilon_1, \varepsilon_1'), \ldots, (\varepsilon_t, \varepsilon_t')\}$: The k-th element of this list is assigned to the point in $P \cup Q$ that is the k-th closest to p_i. For any $\varepsilon > 0$, the disk of radius ε, centered at the k-th closest point, has at most two arcs appearing on $\partial U_{p_i}(\varepsilon)$. An arc of the disk disappears from $\partial U_{p_i}(\varepsilon)$ at $\varepsilon = \varepsilon_k$, and the other arc disappears from $\partial U_{p_i}(\varepsilon)$ at $\varepsilon = \varepsilon_k'$. Since $\mathcal{D}_{p_i}$ is an array of size $m+n$, we can access each element in $O(1)$ time.

Theorem 3. *For two polygonal curves with m and n vertices, the decision problem for the unordered case can be solved in $O(mn)$ time with $O(mn\log(mn))$ preprocessing time. A covering sequence can be computed in the same time.*

4.2 Algorithm for the Optimization Problem

We apply binary search on the set of distances between pairs of points from $P \cup Q$ involved in each step. Without loss of generality, assume that $n \leq m$. There are $O((m+n)^2)$ distances between pairs of points from $P \cup Q$, but we will show that we need only $O(mn)$ of them to compute the optimal distance ε^*.

1. Compute the set $\mathcal{D}$ of distances between pairs of points that are either both from P, 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 above. Let $[\varepsilon_1, \varepsilon_2]$ be the interval returned by the decision algorithm with $\varepsilon_1, \varepsilon_2 \in \mathcal{D}$. If $\varepsilon_1 \neq \varepsilon^*$ and $\varepsilon_2 \neq \varepsilon^*$, then ε^* is the distance of a pair of points in Q.
3. To find ε^*, for each point $p_i \in P$,
   (a) compute the set S_{p_i} of points in $P \cup Q$ that are at distance at most ε_2 from p_i, and construct the Voronoi diagram $\mathrm{VD}(S_{p_i})$.

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

4. For a point pair (p_i, q_j), there exists at most one such point $r \in Q$, thus there are $O(mn)$ candidates in total, and we sort them and again apply binary search on the sorted list with the decision algorithm above.

Analysis. Let (p_i, q_j, r) be a tuple realizing ε^*, i.e., $\max(\|p_i - r\|, \|q_j - r\|) = \varepsilon^*$. Clearly, r is the point in $P \cup Q$ that minimizes $\max(\|p_i - r\|, \|q_j - r\|)$. If $r \in Q$, then r is the point in S_{p_i} that is closest to q_j. Thus, r is the point site associated with the Voronoi cell in $\mathrm{VD}(S_{p_i})$ that contains q_j. 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. Sorting the distances in $\mathcal{D}$ takes $O(mn \log(mn))$ time. The binary search on the sorted list with the decision algorithm takes $O(mn \log(mn))$ time as the preprocessing phase is executed only once for each $p_i \in P$ and the history and deletion lists can be reused for different radii. In step 3, the Voronoi diagram $VD(S_{p_i})$ can be constructed in $O(m \log(mn))$ time for each $p_i \in P$, and the point location can be performed in the same time. Step 3(b) takes $O(m \log(mn))$ time for each $p_i \in P$.

Several Curves. The decision algorithm can be extended to k curves $P^1 = (p_1^1, \ldots, p_{n_1}^1), ..., P^k = (p_1^k, \ldots, p_{n_k}^k)$. If the outer loop iterates over all points $p_{i_1} \in P^1$ for $1 \leq i_1 \leq n_1$, then we determine which points $p_{i_2} \in P^2, \ldots, p_{i_k} \in P^k$ lie inside the disk of radius ε centered at p_{i_1}. For all tuples $(i_1, \ldots, i_k)$ the corresponding entries in the k-dimensional free space matrix are marked as free. The running time is $O(n_1 N \log N + M)$ where $N = \sum_{i=1}^{k} n_i$ and $M = \prod_{i=1}^{k} n_i$. If the history data structure has already been constructed, this running time can be reduced to $O(n_1 N + M)$ time. For a constant $k \geq 2$ curves of size at most n each, the running time becomes $O(n^k)$.

To compute an optimal middle curve, we sort all distances between 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 4. *For two polygonal curves with m and n vertices, the optimization problem for the unordered case can be solved in $O(mn \log(mn))$ time. An optimal covering sequence can be computed in the same time. For $k \geq 2$ curves of size at most n each, the optimization can be solved in $O(n^k \log n)$ time.*

Acknowledgments. This work was initiated at the 17$^{\text{th}}$ 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. Alt, H., Godau, M.: Computing the Fréchet distance between two polygonal curves. Int. J. Comput. Geom. Appl. **5**, 75–91 (1995)
2. Buchin, K., Buchin, M., Gudmundsson, J., Löffler, M., Luo, J.: Detecting commuting patterns by clustering subtrajectories. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 644–655. Springer, Heidelberg (2008)
3. Buchin, K., Buchin, M., van Kreveld, M., Löffler, M., Silveira, R.I., Wenk, C., Wiratma, L.: Median trajectories. Algorithmica **66**(3), 595–614 (2013)
4. Dumitrescu, A., Rote, G.: On the Fréchet distance of a set of curves. In: Proceedings of the 16th Canadian Conference on Computational Geometry, CCCG 2004, Concordia University, Montréal, Québec, Canada, pp. 162–165, 9–11 August 2004
5. Eiter, T., Mannila, H.: Computing discrete Fréchet distance. Technical report, Technische Universität Wien (1994)
6. Har-Peled, S., Raichel, B.: The Fréchet distance revisited and extended. ACM Trans. Algorithms **10**(1), 3:1–3:22 (2014)
7. Sriraghavendra, E., Karthik, K., Bhattacharyya, C.: Fréchet distance based approach for searching online handwritten documents. In: Proceedings of the Ninth International Conference on Document Analysis and Recognition, ICDAR 2007, vol. 1, pp. 461–465. IEEE Computer Society (2007)
8. van Kreveld, M.J., Löffler, M., Staals, F.: Central trajectories. In: 31st European Workshop on Computational Geometry (EuroCG), Book of Abstracts, pp. 129–132 (2015)
9. Zhu, H., Luo, J., Yin, H., Zhou, X., Huang, J.Z., Zhan, F.B.: Mining trajectory corridors using Fréchet distance and meshing grids. In: Zaki, M.J., Yu, J.X., Ravindran, B., Pudi, V. (eds.) PAKDD 2010, Part I. LNCS, vol. 6118, pp. 228–237. Springer, Heidelberg (2010)