Chapter 6

Matching a Polygonal Curve in a Graph of Curves

The Fréchet distance as we have considered it in Chapter 5 is defined for two curves. In this chapter we generalize this concept to one curve and a planar embedded graph with a straight-line embedding. We show how we can find a polygonal curve in the graph that minimizes the Fréchet distance to the other curve. To the best of our knowledge this problem has not been considered before. Apart from the theoretical merit this problem has applications in the construction of road maps from data collected from GPS receivers. See Section 6.1 for more details on this application. The ideas of this chapter have been established in collaboration with Günter Rote, Alon Efrat, and Helmut Alt, see [14]. We thank Scott Howard Morris for introducing us to the application of matching GPS curves, and Lingeshwaran Palaniappan for implementing the algorithm of Section 6.2.

Section 6.2 describes the basic algorithm, which runs in $O(pq\log(pq)\log q)$ time, where q is the complexity of the planar graph and p is the number of line segments of the given polygonal curve. This result is somehow surprising in that it is only one log-factor slower than the computation of the Fréchet distance between two curves, although we have to consider various sub-curves in the given planar graph. Note that the setting we consider is a partial matching variant, since we search for the best matching curve in the graph. However we do not consider any sets of transformations in this chapter. In Section 6.3 we describe how to solve some variants of the basic problem, which include a time-space trade-off. For this we solve the problem for a parameter $1 \leq t \leq \sqrt{p}$ in $O(pq(t+\log q))$ time and $O(qp/t)$ space. Applications dealing with very large graphs, such as the road map construction application, will favor a time-space trade-off due to memory restrictions. Furthermore, the trade-off-algorithm together with the variant of Section 6.3.2 can be used to create a trade-off for the computation of the Fréchet distance for two curves.

6.1 Problem Statement

We consider a given polygonal curve, and a planar embedded graph with line segment edges, and we wish to find a path in the graph (which then corresponds to a polygonal curve) such that the Fréchet distance between the curve and the path is minimized. This is a partial matching variant which is motivated by the following application: The Global Positioning System (GPS) is a collection of satellites that provides worldwide positioning information. A specific position can be determined by using a GPS receiver. Now consider a given road map, and a person traveling on some of the roads, while recording its information using a GPS receiver. The road map can be modeled by a planar embedded graph, and the path the person traveled is represented by a sequence of *GPS positions* recorded by the GPS receiver, which we connect by straight line segments to form a polygonal curve. Since the GPS receiver usually introduces noise, the captured curve will not exactly lie on the road map. The task is to identify those roads which have actually been traveled. This is a prerequisite for incrementally constructing road maps from such GPS curves, which is especially interesting for roads such as hiking trails in a forest which are not visible on aerial pictures.

Let us formalize the setting we consider: Let $G = (V, E)$ be an undirected connected planar graph with a given embedding in $\mathbb{R}^2$, $|V| = q$, $|E| = O(q)$, such that $V = \{1, \ldots, q\}$ corresponds to points $\{v_1, \ldots, v_q\}$ in $\mathbb{R}^2$. We assume, although G is an undirected graph, that each undirected edge between vertices $i, j \in V$ is represented by the two directed edges $(i, j), (j, i) \in E$. Thus E consists of directed edges, but still represents an undirected graph. Each edge $(i, j) \in E$ is embedded as an oriented straight line segment $s_{i,j}$ from v_i to v_j. $s_{j,i}$ is obtained from $s_{i,j}$ by reversing the orientation. Furthermore let $\alpha : [0, p] \to \mathbb{R}^2$ be a polygonal curve in $\mathbb{R}^2$, which consists of p line segments $\overline{\alpha}_i := \alpha|_{[i,i+1]}$ for $i \in \{0, 1, ..., p-1\}$. We consider each line segment $\overline{\alpha}_i$ to be parameterized by its *natural parameterization*, i.e., $\alpha(i+\lambda) = (1-\lambda)\alpha(i)+\lambda\alpha(i+1)$ for all $\lambda \in [0, 1]$. For a vertex $i \in V$ we denote by $\mathrm{Adj}(i) \subseteq V$ the set of vertices adjacent to i. A path $\pi = (i_1, \ldots, i_k)$ of length $k - 1$ in G is a sequence of vertices $i_1, \ldots, i_k \in V$ such that $(i_\nu, i_{\nu+1}) \in E$ for all $1 \leq \nu < k$. We denote by β_π the polygonal curve that we obtain by concatenating the line segments $s_{i_1,i_2}, s_{i_2,i_3}, \ldots, s_{i_{k-1},i_k}$ associated with the edges that π visits in the order they occur on π.

We interpret G as a given road map, with given junction vertices where several streets join, and transit vertices that lie in the middle or at a dead end of a street. We assume that α has been obtained from traveling some of the streets in a certain order, however containing some extent of noise stemming from the GPS receiver. Given α and G we wish to find the roads in G that α traveled. We formalize the task as follows:

Problem 8 (Find-Path) *We wish to find a path π in G which minimizes $\delta_{\mathrm{F}}(\alpha, \beta_\pi)$, where β_π is the concatenation of line segments on π.*

Notice that in the most general setting π does neither have to be simple, nor edge-disjoint, since in the road map application it should be allowed to travel in loops and also to travel the same street multiple times. Also we assume that the path starts and ends in a vertex of G, and not in the middle of an edge. However we will see in Section 6.3.1 that this assumption can easily be relaxed.

We attack Problem 8 by first solving the decision problem for which we fix $\varepsilon > 0$ and wish to find a path (if it exists) in G such that the Fréchet distance is at most ε. We refer to this problem as Problem 9. In fact most of Section 6.2 will concentrate on solving this task. Afterwards we apply parametric search, similar as in [10], to eventually solve Problem 8.

Problem 9 (Find-ε-Path) *Let $\varepsilon > 0$ be given. We wish to find a path π in G such that for the concatenation β_π of line segments on π $\delta_{\mathrm{F}}(\alpha, \beta_\pi) \leq \varepsilon$ holds.*

As a subproblem we will consider the decision variant of Problem 9. Note that in this case we do not fix another parameter, but we distinguish between only deciding if there exists a path in G with the desired properties, and actually computing such a path.

Problem 10 (Decide-ε-Path) *Let $\varepsilon > 0$ be given. We wish to decide if there exists a path π in G such that for the concatenation β_π of line segments on π $\delta_{\mathrm{F}}(\alpha, \beta_\pi) \le \varepsilon$ holds.*

6.2 Algorithm

6.2.1 Basic Concepts and Overview

If not stated otherwise let $\varepsilon > 0$ be given. Let $s_{i,j}$ for all $(i,j) \in E$ be continuously parameterized by values in $[0,1]$, thus $s_{i,j} : [0,1] \to \mathbb{R}^2$.

Definition 29 ($F_{i,j}$, F_i, $FD_{i,j}$, FD_i) *Let $i \in V$ and $(i,j) \in E$. Then we define the following free spaces and free space diagrams:*

- $F_{i,j} := F_\varepsilon(\alpha, s_{i,j})$,
- $F_j := F_\varepsilon(\alpha, v_j)$,
- $FD_{i,j} := FD_\varepsilon(\alpha, s_{i,j})$,
- $FD_j := FD_\varepsilon(\alpha, v_j)$.

Furthermore let $\mathbf{L}_j$ be the left endpoint and $\mathbf{R}_j$ be the right endpoint of FD_j.

Compare Definition 17 for the definition of the free space and the free space diagrams. Note that FD_j is a one-dimensional free space diagram consisting of at most $2p+1$ black or white intervals, and F_j is the corresponding one-dimensional free space, which consists of a collection of white intervals. As shown in Lemma 11 (see also [10]) $FD_{i,j}$ consists of a row of p cells. Each such cell corresponds to a line segment of α, and the free space in each cell is the intersection of an elliptical disk with that cell.

For each $i \in V$ the free space diagrams $FD_{i,j}$ and $FD_{j,i}$ for all $j \in \mathrm{Adj}(i)$ have the one-dimensional free space diagram FD_i in common - as the bottom of $FD_{i,j}$ and as the top of $FD_{j,i}$. Thus we can imagine glueing together the two-dimensional free space diagrams along the one-dimensional free space they have in common, according to the adjacency information of G. In this manner we obtain a topological structure which we call the *free space surface* for G and α, see Figure 6.1 for an example.

The algorithm in [10] computes a monotone feasible path in the free space diagram of two polygonal curves in a dynamic programming fashion: Since the part of the free space inside a cell is convex it suffices to consider only the free space on all left boundaries $L^\varepsilon_{i,j}$ and all bottom boundaries $B^\varepsilon_{i,j}$ of the cells, where i and j range between 0 and the number of vertices on the two curves, respectively. The algorithm starts in the lower left corner of the diagram which, assuming a feasible path exists, is white. Otherwise the algorithm stops immediately. Then it computes which parts of the boundary of the incident cell can be reached from the lower left corner by a feasible monotone path. In general this type of reachability information is propagated incrementally on the cell boundaries throughout the diagram. In the end, if the upper right corner is reachable, a monotone path between the lower left and the upper right corner can be reconstructed; otherwise there exists no monotone feasible path.

We apply a related approach to our more general setting: We search for a feasible path in the free space surface. This path has to start at some white left corner $\mathbf{L}_k$ and has to end at some white right corner $\mathbf{R}_j$, for two vertices $j, k \in V$, since the corresponding path π in G has

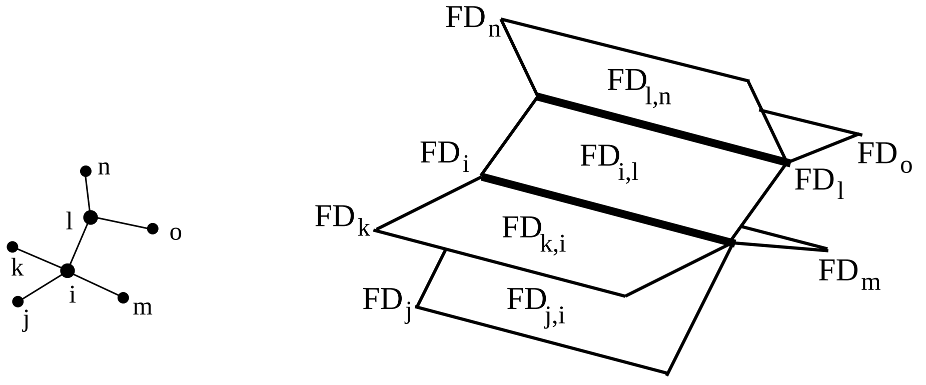


Figure 6.1: Example of a free space surface: Free space diagrams glued together according to the adjacency information of G.

to start and end in a vertex of G. Any path π in G selects a sequence of free space diagrams in the free space surface, whose concatenation yields $FD_{\varepsilon}(\alpha, \beta_{\pi})$. Thus let us consider the following reachability information:

Definition 30 (Reachable points, $\mathcal{R}(j)$) *For every vertex $j \in V$ let $\mathcal{R}(j)$ be the set of all points $u \in F_j$ for which there exists a $k \in V$ and a path π from k to j in G such that there exists a monotone feasible path from $\mathbf{L}_k$ to u in $F_{\varepsilon}(\alpha, \beta_{\pi})$.*

We call points in $\mathcal{R}(j)$ reachable. *We call an interval of points in $\mathcal{R}(j)$* reachable *if every point in it is reachable.*

The definition of reachable points directly yields the following property:

Observation 3 *There is a path π in G with $\delta_{\mathrm{F}}(\alpha, \beta_{\pi}) \leq \varepsilon$ iff there is a vertex $j \in V$ such that $\mathbf{R}_j \in \mathcal{R}(j)$.*

Similar to [10] we first decide whether there exists an ε-path, i.e., solve Problem 10, by computing $\mathcal{R}(j)$ for all $j \in V$ in a dynamic programming manner. In fact we will not store the whole $\mathcal{R}(j)$ but only parts of it which allow us to arrive at the correct decision. However as a preprocessing step we first compute the free space diagrams $FD_{i,j}$ together with some additional reachability information. Altogether the algorithm solving Problem 9 consists of three stages: The preprocessing stage (which we describe in Section 6.2.2), the dynamic programming stage (see Section 6.2.3) which solves Problem 10, and finally the path reconstruction stage (see Section 6.2.4) which solves Problem 9 by constructing the path π in G along with feasible reparameterizations of β_{π} and α that witness the fact that $\delta_{\mathrm{F}}(\alpha, \beta_{\pi}) \leq \varepsilon$. In Section 6.2.6 we show how to apply parametric search to solve Problem 8.

In the following we make use of a property of $FD_{i,j}$ for each $(i, j) \in E$, which we call the *simplicity property* of $FD_{i,j}$: Each $FD_{i,j}$ is a row of cells, and each white region in such a cell is the intersection of an elliptical disk with the cell. Thus there is no vertical line at any position in $FD_{i,j}$ which contains white, black, and white points alternatingly. Or in other words, the white points on a vertical line always form an interval. From this we obtain the following insight:

Lemma 25 *Let $(i,j) \in E$, and $u \in F_i$, $v \in F_j$ be white points with $u \le v$ for which a feasible monotone path exists from u to v in $FD_{i,j}$. Then for every $u' \in F_i$ and $v' \in F_j$, $u \le u' \le v' \le v$ there exists a feasible monotone path in $FD_{i,j}$ from u' to v'.*

Proof: Consider the feasible monotone path from u to v. Then due to the simplicity property of $FD_{i,j}$ it is possible to go straight upward from u' until hitting this path, and similarly to go straight downward from v' until hitting this path, and stay inside the free space all the time. Stitching those pieces of paths together we obtain the desired feasible monotone path in $FD_{i,j}$ from u' to v'. $\square$

6.2.2 Preprocessing

We compute all one-dimensional free space diagrams FD_i for all $i \in V$. Conceptually we continue to consider the $FD_{i,j}$ for all $(i,j) \in E$, but we do not need to compute them explicitly, for we capture the reachability information in the additional pointers we will compute. Let $(i,j) \in E$ be fixed, then $FD_{i,j} \subseteq [0,p] \times [0,1]$ consists of p cells, one for each

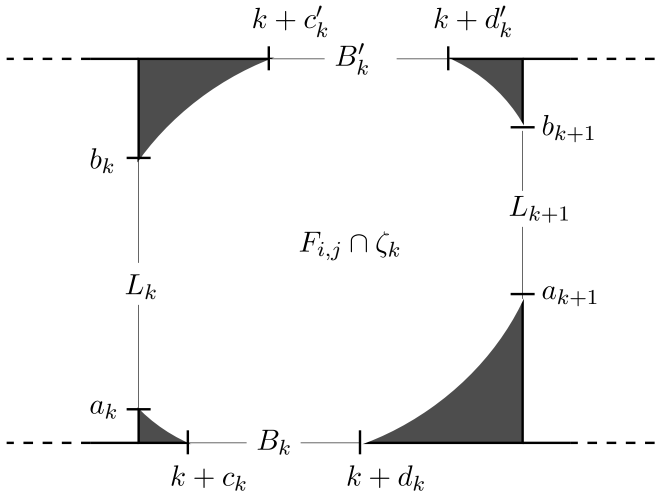


Figure 6.2: Intervals of the free space on the boundary of a cell.

segment in α. Let, similar to Definition 19, ζ_k be the cell in $FD_{i,j}$ corresponding to the kth segment $\overline{\alpha}_k$ of α, $0 \le k \le p-1$. Let $L_k = [a_k, b_k]$ be the white interval on the left boundary of ζ_k, $B_k = [k + c_k, k + d_k]$ be the white interval on the bottom boundary of ζ_k, $B'_k = [k + c'_k, k + d'_k]$ be the white interval on the top boundary of ζ_k. See Figure 6.2 for an illustration. If $L_k = \emptyset$ then we set $a_k := 1$ and $b_k := 0$. Similarly if $B_k = \emptyset$ we set $c_k := 1$ and $d_k := 0$, and if $B'_k = \emptyset$ we set $c'_k := 1$ and $d'_k := 0$. Note that the left boundary of ζ_k is part of the vertical line segment $\{k\} \times [0,1]$ with respect to the free space diagram $FD_{i,j}$. We call $\{k\} \times \mathbb{R}$ the *vertical line at k*. We call k the *index* of the two spikes bounding L_k. Note that the interval endpoints correspond to heights or widths of spikes.

Definition 31 (l-Pointers and r-Pointers) *Let $(i,j) \in E$, and I be a white interval of FD_i. Then we denote by $l_{i,j}(I)$ the leftmost point (and by $r_{i,j}(I)$ the rightmost point) on*

FD_j which can be reached from some point in I by a monotone feasible path in $FD_{i,j}$. We call $l_{i,j}(I)$ a left pointer *or l-*pointer *and $r_{i,j}(I)$ a* right pointer *or r-*pointer*.*

For all $(i,j) \in E$ we compute $l_{i,j}(I)$ and $l_{i,j}(I)$ for each white interval I of FD_i. This can be done in linear time for all intervals on FD_j, see Lemma 28. Note that $l_{i,j}(I)$ either equals the left endpoint of I or equals $k + c'_k$ for some $0 \le k \le p-1$. For the right pointer holds $r_{i,j}(I) = k + d'_k$ for some other $0 \le k \le p-1$. Note that similar reachability pointers have been used in [10] for attacking the case of closed curves. Let us call $l(I)$ the left endpoint of I, and $r(I)$ the right endpoint of I.

For notation purposes we identify in the following a white interval I on FD_i with a B_k for a $0 \le k \le p-1$. If a white interval on FD_i spans several cells we consider it to be composed of one white interval per cell.

For each white interval I of FD_i we store the left pointers and right pointers in two arrays that are indexed by the $j \in \mathrm{Adj}(i)$. Thus each white interval I on FD_i has $|\mathrm{Adj}(i)|$ l-pointers and r-pointers attached to it.

Lemma 26 *Let $(i,j) \in E$. Then for each white interval I on FD_i holds $l_{i,j}(I) = l_{i,j}(l(I))$ and $r_{i,j}(I) = r_{i,j}(l(I))$.*

Proof: Let $u \in I$ such that $l_{i,j}(I) = l_{i,j}(u)$. Since I is white, the horizontal line segment from $l(I)$ to u is also white, such that $l_{i,j}(u)$ can also be reached by a monotone feasible path from $l(I)$. The same argument holds for $r_{i,j}$. □

The following lemma characterizes when points on FD_j can be reached from points on FD_i by a monotone feasible path in $FD_{i,j}$. We use Lemma 27 in the proof of Lemma 28, in which we show how to compute all l-pointers and r-pointers.

Lemma 27 *Let $(i,j) \in E$ be fixed. Let $0 \le k < k' \le p-1$, and assume that $B_k, B'_{k'} \ne \emptyset$. Then there is a monotone feasible path in $FD_{i,j}$ from some point on B_k to some point on $B'_{k'}$ if and only if*

$$\max_{i=k+1}^{l} a_i \le \min_{i=l}^{k'} b_i \quad \textit{for all} \quad k < l \le k'. \tag{6.1}$$

Proof: Assume there is a monotone path π in $F_{i,j}$ from a point on B_k to a point on $B'_{k'}$. For each $k < l \le k'$ consider the point where π passes the vertical line at l. π has to pass above all a_i for $i = k+1, \ldots, l$ and below all b_j for $j = l, \ldots, k'$, otherwise it would not be a monotone feasible path.

For the other direction, assume that (6.1) holds for all $k < l \le k'$. Let $a_{i_1}, \ldots, a_{i_m}$ be the sequence of different indices that form the partial maxima of the sequence $a_1, \ldots, a_{p-1}$, when considering its prefixes obtained by reading it from left to right. We construct π to start in an arbitrary point on B_k, go vertically upwards until the height a_{i_1}, go horizontally until we hit the lower spike in i_1, then visit the points $a_{i_1}, \ldots, a_{i_m}$, and then pass horizontally until it ends below some point on $B'_{k'}$, which it then connects to by going vertically straight up. Two points a_{i_ν} and $a_{i_{\nu+1}}$ are connected in π by a path that starts horizontally at height a_{i_ν} until it hits the lower spike in $i_{\nu+1}$. It then follows the boundary of this spike (which is monotonically increasing) until the height $a_{i_{\nu+1}}$. By construction π is monotone. Since (6.1) holds for $l = 1, i_1, \ldots, i_m$ each described piece in the path is indeed feasible. □

Lemma 28 *Let $(i, j) \in E$. Then all pointers $l_{i,j}(B_k)$ and $r_{i,j}(B_k)$ for all white intervals B_k on FD_i, $1 \leq k \leq p-1$, can be computed in $O(p)$ time.*

Proof: The left pointers $l_{i,j}(B_k)$ for all $0 \leq k \leq p-1$ are easily computed by a scan for increasing $k = 0, \ldots, p-1$: Let k be fixed. If $c_k \leq d'_k$ then we set $l_{i,j}(B_k) := k + \max(c_k, c'_k)$. Otherwise we greedily search for the first cell $\zeta_{k'}$, $k' > k$, which contains a white point on its upper boundary, and such that (6.1) holds. If such a cell does not exist then B_k is black. Otherwise we set $l_{i,j}(B_k) := k' + c'_{k'}$. For the next iteration, i.e., for k increased by one, we only have to consider cells to the right of $\zeta_{k'}$, such that in total we visit every cell at most once.

The computation of the right pointers is slightly more complicated. We proceed incrementally for $k = 0, \ldots, p-1$ as follows. For each k, if $B_k \neq \emptyset$, we compute the largest value k' for which (6.1) holds. In order to do this we maintain a stack $\mathcal{S} := \{i_1, \ldots, i_m\}$ of indices $k < i_1 < i_2 < \cdots < i_m \leq k'$ which are the indices of those lower spikes that are horizontally visible from the vertical line at k'. In other words, $\mathcal{S}$ is the sequence of different indices that form the partial maxima of the sequence $a_{k+1}, \ldots, a_{k'}$, when reading it from left to right. Thus each index $i_s \in \mathcal{S}$ is characterized by the property that $a_{i_s} > a_l$ for all $i_s < l \leq k'$. We call $\mathcal{S}$ the *partial maxima stack*, with *top* element i_m, and *bottom* element i_1. Note that for $\mathcal{S} = \{i_1, i_2, \cdots, i_m\}$ we have $i_1 < i_2 < \cdots < i_m$ and $a_{i_1} > a_{i_2} > \cdots > a_{i_m}$. See Figure 6.3 for an illustration. The significance of these values is as follows: Let $i_s < i_{s+1} \in \mathcal{S}$ be two successive indices, and let $i_s < i \leq i_{s+1}$. Then the lowest point on the vertical line at k' that can be reached from B_i (if $B_i \neq \emptyset$) by a monotone feasible path in $FD_{i,j}$ is $a_{i_{s+1}}$.

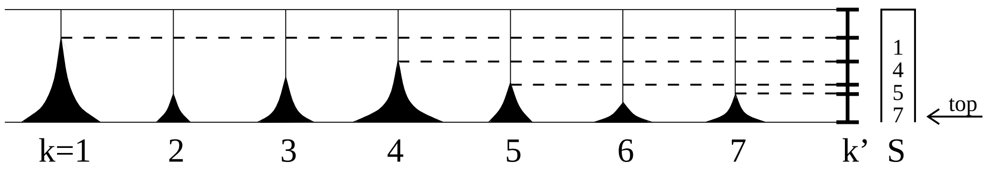


Figure 6.3: An example of lower spikes and their partial maxima stack $\mathcal{S}$.

We initialize $\mathcal{S} = \{0\}$ and $k' = 1$. Let $k = 0, \ldots, p-1$ be the current value of the iteration. We maintain the invariant that (6.1) holds for the current values of k and k' throughout the algorithm. This is trivially true for the initialization case. And if we know that (6.1) holds for $k-1$ and k', then it immediately holds for k and k'. For fixed k we now search for the maximal k' that fulfills (6.1). (We always denote the top element of $\mathcal{S}$ by a_{i_1} and the bottom element by a_{i_m}, although the indices and the value of m change during the algorithm.) If $a_{i_1} > b_{k'+1}$, then $k'+1$ violates (6.1), thus k' is the maximal value we searched for. If $a_{i_1} \leq b_{k'+1}$, then we have $\max\{a_{i_1}, a_{k'+1}\} = \max_{i=k+1}^{k'+1} a_i \leq b_{k'+1}$, thus (6.1) holds for $k'+1$ and we can safely increase k' by one. Now we have to maintain $\mathcal{S}$ to represent the partial maxima of lower spikes between k and the increased value k'. For this we pop the topmost values from $\mathcal{S}$ until $a_{i_m} > a_{k'}$. Finally we push k' on top. Then we start with a new iteration on k'.

Once we have found the maximal k' that fulfills (6.1), we know that there is no monotone feasible path in $FD_{i,j}$ from any point on B_k (assuming that $B_k \neq \emptyset$) to $B'_{k'+1}$. Thus the rightmost point on FD_j that can be reached by a monotone feasible path from B_k is the first d'_w which bounds a white interval on FD_j to the left of the vertical line at $k'+1$.

In order to obtain all d'_w efficiently during the run of the algorithm we store $O(p)$ *shortcut* *pointers* for each FD_i: At the *givesak*-th cell boundary of FD_i, for integer $0 \leq k \leq p-1$, we store a pointer to the rightmost white point on FD_i that lies to the left of k. If there is no such white point we set the shortcut pointer to NIL. We construct this pointer structure on the fly by computing a pointer value from the shortcut pointer to its left. Now we find d'_w by greedily searching for the next white point on FD_j to the left of $k'+1$. If possible we follow the next shortcut pointer; otherwise we greedily search for the first white point and compute the shortcut pointers on the way until we either hit an already computed shortcut pointer or the end of FD_j. If $k < w$ then we set $r_{i,j}(B_k) := w + d'_w$. If $k > w$ then we set B_k to be black. If $k = w$ then if $c_k \leq d'_k$ we set $r_{i,j}(B_k) := k + d'_k$, otherwise we set B_k to be black.

Finally, if $i_1 = k+1$ then we remove i_1, i.e., the bottommost element, from $\mathcal{S}$. Then we start the next iteration on k with its value increased by one.

For the runtime analysis, note that k and k' are always increased, and never decreased. In each such increasing step we perform only constant time operations without counting the stack operations and the location of the d'_w. Once a value is removed from the stack (either by popping from the top, or by removing from the bottom) it is never inserted in $\mathcal{S}$ again. Thus every integer between 1 and $p-1$ is at most once inserted in the stack or removed from the stack. With respect to the shortcut pointers we charge every cell boundary for computing its shortcut pointer. Thus the total time to compute all $r_{i,j}(I)$ is indeed $O(p)$.

□

6.2.3 Dynamic Programming

In this stage we decide if there exists a feasible monotone path in the free space surface. Thus we attack Problem 10. Note that such a path traverses a sequence of free space diagrams $FD_{i,j}$. We call the part of a path that traverses one such free space diagram a *segment* of the path.

Conceptually we sweep all $FD_{i,j}$ at once with a vertical sweep line from left to right. Let $0 \leq x \leq p$ denote the position of the sweep line. For each $i \in V$ we will store a set $C_i \subseteq \mathcal{R}(i) \subseteq F_i$ of white points, which we compute in a dynamic programming manner. Throughout the algorithm we maintain the following invariant:

Definition 32 (C_i) *Let $i \in V$ and x be any current position of the sweep line. Then C_i consists of all reachable points $u \in \mathcal{R}(i) \subseteq F_i$, such that $u \geq x$, and for which the last segment of their associated feasible monotone path crosses or ends at the sweep line.*

Thus we are able to decide if $\mathbf{R}_i \in \mathcal{R}(i)$ by checking if $\mathbf{R}_i \in C_i$ for an advanced enough position x of the sweep line.

Definition 33 (Consecutive Chain) *The subsequence of all white intervals of a sequence of consecutive white and black intervals of FD_i is called a* consecutive chain *of intervals. For a consecutive chain C let $l(C)$ be its left and $r(C)$ be its right endpoint. We also use the same notation for single intervals. For two consecutive chains $C' \subseteq C$ we call C' a* consecutive subchain *of C.*

Lemma 29 *Every C_i, for $i \in V$, is a consecutive chain, for every value of x. Therefore $C_i = [l(C_i), r(C_i)] \cap FD_i$.*

Proof: Let x and let $i \in V$ be fixed. Let $w \in C_i$ be the largest point in C_i. By definition of C_i there is a $j \in \text{Adj}(i)$ and a white point $u \in F_j$ with $u \leq x \leq w$, such that u is reachable and there exists a monotone feasible path in $FD_{j,i}$ from u to w. For any white point $v \in F_i$ with $x \leq v \leq w$ there exists by Lemma 25 a monotone feasible path from u to v in $FD_{j,i}$, which makes v in particular also reachable by the same path that reaches u, concatenated by the monotone feasible path from u to v. Thus $v \in C_i$, and C_i is a consecutive chain.

From the definition of a consecutive chain follows that $C_i = [l(C_i), r(C_i)] \cap FD_i$. □

The algorithm we present is a mixture of a sweep (since we are sweeping with a sweep line), dynamic programming (on the C_i we incrementally build up), and Dijkstra's algorithm for shortest paths (since we are computing paths using a priority queue to augment the path in a similar fashion to Dijkstra's algorithm). We maintain a priority queue Q of white intervals of FD_i which are known to be reachable. More precisely, for each $i \in V$ the first white interval of C_i (if $C_i \neq \emptyset$) is stored in Q. The priority of an interval is its left endpoint. The events for the sweep line, i.e., the different values of x, are the left endpoints of the intervals in Q. Every interval in Q is part of a consecutive chain to which we store a pointer with the interval. Since $C_i = [l(C_i), r(C_i)] \cap FD_i$ we store the C_i implicitly in constant space by storing only $l(C_i)$ and $r(C_i)$.

We initialize Q with all white $\mathbf{L}_i$ (which are degenerate intervals). For all $i \in V$ if $\mathbf{L}_i$ is white we set $C_i := \mathbf{L}_i$, otherwise $C_i := \emptyset$. Then we process these intervals in increasing order as follows:

1. Extract and delete the leftmost interval I from Q; if there are several intervals with the same priority pick an arbitrary one. Advance x to $l(I)$.

2. Let C_i be the consecutive chain that contains I. Insert the next white interval of C_i which lies to the right of I, into Q.

3. For each $j \in \text{Adj}(i)$ update C_j to comply with the new value of x. If necessary, update the interval of C_j in Q.

4. Store for each white interval J that has been newly added to C_j (or that has been enlarged) a *path pointer* to the interval I (from which it can be reached by a monotone feasible path in $FD_{i,j}$.

We process all intervals in Q until we either find a $j \in V$ such that $\mathbf{R}_j \in C_j$, or until Q is empty. In the latter case there is no path in G with $\delta_{\text{F}}(\alpha, \beta_\pi) \leq \varepsilon$. In the first case we know that there exists such a path, and we reconstruct it using the path pointers in the second stage of the algorithm, which is described in Section 6.2.4.

The proof of Lemma 30 gives more details about step 3 of the algorithm.

Lemma 30 *Let x be the current position of the sweep line, and let C_j, for $j \in Adj(i)$, be the consecutive chain stored at vertex j for the current value of x. Let $I \in C_i$ be the next interval in Q. Then C_j can be updated in constant time to comply with the new position $l(I)$ of the sweep line. The update of the first interval of C_j in Q takes $O(\log p)$ time.*

Proof: $[l_{i,j}(I), r_{i,j}(I)]$ defines a consecutive chain on FD_j, whose white intervals are white intervals on FD_j which have now been identified to be reachable, since all their points can be reached from points on I (which are all reachable) by a feasible monotone path in $FD_{i,j}$.

Each such white interval is either a complete white interval of FD_j, or a suffix of a white interval of FD_j. By Lemma 26 we know that $[l_{i,j}(I), r_{i,j}(I)] = [l_{i,j}(l(I)), r_{i,j}(l(I))]$, i.e., we have to update C_j to comply with the new position $l(I)$ of the sweep line by including the white point of $[l_{i,j}(I), r_{i,j}(I)]$ into C_j.

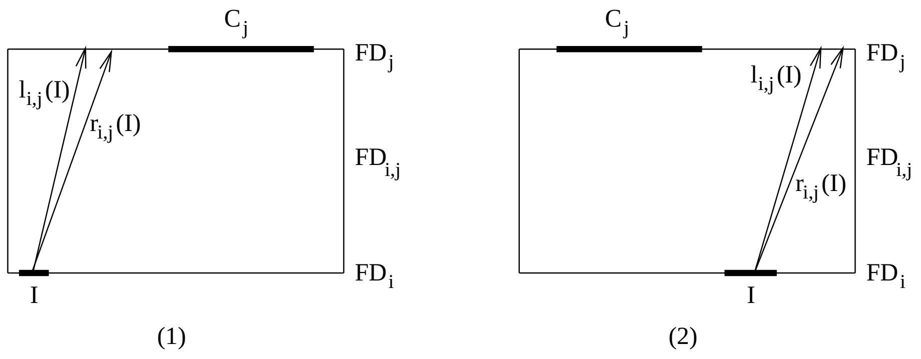


Figure 6.4: Merging of consecutive chains.

- If $C_j = \emptyset$ then we set $C_j := [l_{i,j}(I), r_{i,j}(I)] \cap FD_j$. We insert the first white interval of C_j into Q.
- If $C_j \neq \emptyset$ there arise two different cases, (1) $l(I) \leq r(C_j)$, and (2) $l(I) > r(C_j)$, see Figure 6.4.

  In case (1) the new C_j equals $([l_{i,j}(I), r_{i,j}(I)] \cup C_j) \cap FD_j = [\min(l_{i,j}(I), l(C_j)), \max(r_{i,j}(I), r(C_j))] \cap FD_j$. Note that by Lemma 29 this is again a consecutive chain. If $l_{i,j}(I) < l(C_j)$ then we delete the interval of the old C_j that is contained in Q, and insert the first white interval of the new C_j into Q.

  In case (2) we set $C_j := [l_{i,j}(I), r_{i,j}(I)] \cap FD_j$, since the new C_j contains only points to the right of $l(I)$. We delete the interval of the old C_j that is contained in Q, and insert the first white interval of the new C_j into Q.

Clearly each update of C_j itself takes constant time. Assuming an appropriate implementation of the priority queue, each operation on Q takes $O(\log p)$ time. □

The path pointer for a newly added white interval J in step 4 of the algorithm is stored as follows: Each reachable interval is a suffix of a white interval of FD_j. Thus we can store the path pointer at the right endpoint of J in F_j, pointing to the right endpoint of I in FD_i. It might also happen that J was already present in C_j, but its left endpoint has been changed to x. This means that its left endpoint is smaller than it was before, i.e., J has been enlarged. In this case we delete the old path pointer at J and replace it by the new path pointer to I. Also, since J is present in Q, we update its left endpoint, and with that its priority, in Q.

6.2.4 Path Reconstruction

We assume that in the dynamic programming stage we found a $j \in V$ with $\mathbf{R}_j \in J$, where J is a white interval in C_j for some position x of the sweep line. In this stage we use the path pointers to construct a path π in G together with a feasible monotone path in $FD_\varepsilon(\alpha, \beta_\pi)$ which witnesses the fact that $\delta_F(\alpha, \beta_\pi) \leq \varepsilon$.

By construction the right endpoint of J has a path pointer attached to it. We follow this path pointer to the right endpoint of an interval I, which is a suffix of an interval on FD_i for an $i \in \mathrm{Adj}(j)$. We repeat following the path pointers until we end at an $\mathbf{L}_k$. This way we obtain a sequence of pairs (i, r) where $i \in V$ and r is the right endpoint of the visited interval on FD_i. We call this sequence the *path sequence*. Note that it starts with $(k, \mathbf{L}_k)$ for a $k \in V$. When we extract the first component of each pair, we obtain a sequence of $i \in V$ that represents the desired path π in G. It remains to construct the corresponding feasible monotone path in $FD_\varepsilon(\alpha, \beta_\pi)$. Note that for two successive pairs (i, r) and (i', r') in the path sequence it can indeed be that $r' < r$. In order to construct the feasible monotone path described by the path sequence we use a partial minima stack, this time on the right spikes. This stack is defined similarly to the partial maxima stack of Lemma 27. We scan the path sequence from beginning to end, use the partial minima stack to *snap* the right endpoints to the respective minimum endpoint of the prefix of the path sequence processed so far. This way we obtain a monotone increasing sequence of the right endpoints of this *modified* path sequence. Note that this snapping process cannot cause a right endpoint to be snapped to the left of the left endpoint of its underlying interval. This follows from the order of processing the intervals in Q, and from storing a path pointer for a point on FD_i to the first interval that made this point reachable. Thus the modified path sequence is indeed still a valid path sequence encoding a monotone feasible path. Now for each successive pair in the modified path sequence we scan the row of cells in the corresponding free space from left to right, and construct a monotone path along the heights of the lower spikes (using a partial maxima stack again to keep the path monotone). By concatenating these paths we obtain a desired monotone feasible path witnessing the fact that $\delta_F(\alpha, \beta_\pi) \leq \varepsilon$.

6.2.5 Time Analysis

Theorem 11 *The described algorithm solves Problem 10, i.e., it decides if there is a path π in G such that $\delta_F(\alpha, \beta_\pi) \leq \varepsilon$ in $O(pq \log q)$ time and $O(pq)$ space. If such a path π exists the algorithm solves Problem 9 by computing π together with a monotone feasible monotone path in the free space surface, in $O(pq \log q)$ time and $O(pq)$ space.*

Proof: Each FD_i has complexity $O(p)$ and can be constructed in $O(p)$ time. Each interval I on FD_i has $|\mathrm{Adj}(i)|$ l-pointers and r-pointers attached to it. The number of all l- and r-pointers for all FD_i sums up to $O(p|E|) = O(pq)$, and can by Lemma 28 also be constructed in this time. Thus we need $O(pq)$ time and space for the preprocessing.

In the dynamic programming stage we insert and delete a suffix of every white interval of any FD_i, $i \in V$, at most once in Q. Also the left endpoint of a white interval of any FD_i might be changed $|\mathrm{Adj}(i)|$ times. $|Q| \leq q$ since Q always contains at most one interval per vertex in G. With an appropriate implementation of a priority queue each operation needs $O(\log q)$ time, thus $O(pq \log q)$ altogether.

For each interval in Q we consider each j in the adjacency list of its consecutive chain and spend constant time to merge consecutive chains and construct path pointers for each such j. Altogether this sums up to a runtime of $O(p|E|) = O(pq)$, and for the whole dynamic programming stage together with the priority queue operations to $O(pq \log q)$.

Note that we only store one consecutive chain per vertex, and Q contains at most one interval per vertex. Thus the extra storage for all the consecutive chains in the dynamic programming stage is only $O(q)$. However we store one path pointer per interval in FD_i, thus the space complexity for the path pointers is $O(pq)$.

For an J which has been identified to be reachable we construct a path pointer to the interval I that makes J reachable, which means that $l(I) \leq l(J)$. Since we process intervals by increasing left endpoint we know that there cannot be a cycle in the path pointers. Thus every path pointer can be contained in a monotone feasible path in the free space surface at most once. Since there are $O(pq)$ path pointers, the worst-case length of a feasible path in the free space surface and thus also of a path π in G is $O(pq)$. This is tight, as a construction of α zigzagging horizontally, and G representing a vertically zigzagging curve shows. Since we need time linear in the length of the path to reconstruct it, the path reconstruction needs $O(pq)$ time at no extra storage. □

Note that the space of $O(pq)$ is needed only in two places, namely in the preprocessing stage to compute the FD_i together with all l-pointers, r-pointers, and shortcut pointers, and also in the dynamic programming stage to store the path pointers. However for the GPS application there is a huge amount of road map data, which forbids a quadratic space algorithm since all the free space data could not even fit into memory. We show in Section 6.3.3 how our algorithm can be tuned to trade space for runtime, which in this case is desirable. For this we do not explicitly pre-compute the reachability information, but we will see that instead we can compute it on the fly during the dynamic programming algorithm. For reducing space for the path pointers, we will apply a standard space-saving technique for dynamic programming [52, 50] in the path reconstruction stage.

6.2.6 Parametric Search

Now let us turn to solve the original problem Problem 8. Analogously to [10] we use the algorithm which solves Problem 9 together with Megiddo's parametric search technique [62] to find the optimal ε. We will in fact also apply Cole's trick for parametric search based on sorting such that we obtain only a logarithmic instead of a poly-logarithmic overhead.

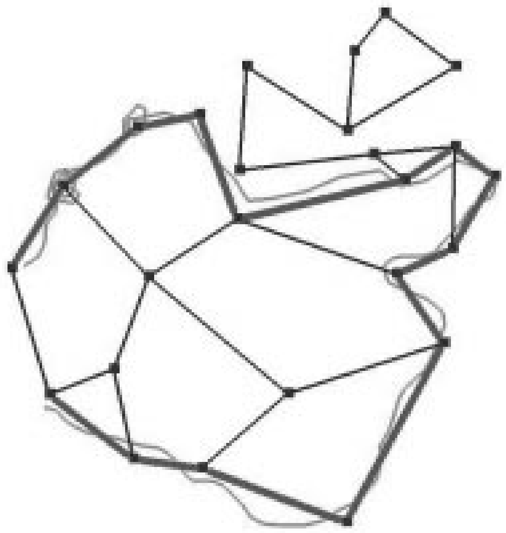
Note that the outcome of the algorithm which solves Problem 9 depends solely on the relative positions of all possible widths and heights of spikes in all free space diagrams in the free space surface. For varying ε all those values depend on ε, and for the parametric search an ε is *critical* if it makes two of these widths or heights coincide. There are $O(pq)$ different widths or heights of spikes. As in [10] we now apply a parallel sorting algorithm on those $O(pq)$ values which depend on ε, and generate in that way a superset of the critical values of ε we need. By utilizing Cole's trick [39] for parametric search based on sorting, which in general yields a runtime of $O((k + T_{seq}) \log k)$ where T_{seq} is the sequential runtime for the decision problem and k is the number of values to be sorted, we thus obtain a runtime of $O(pq \log(pq) \log q)$, at no extra storage.

Theorem 12 *There is an algorithm that solves Problem 8, i.e., it finds a path π in G which minimizes $\delta_{\mathrm{F}}(\alpha, \beta_\pi)$, in $O(pq \log(pq) \log q)$ time and $O(pq)$ space.*

Notice that recently van Oostrum et al. [70] have implemented an object-oriented framework in order to simplify the usually rather complicated implementation of parametric search. See our remark on that at the end of Section 3.1.1. Interestingly they demonstrated their environment on the computation of the Fréchet distance, and showed that their framework which uses quicksort, instead of a parallel sorting algorithm or Cole's trick, runs faster in practice than the usually recommended binary search on the number space of ε. Therefore it might be worthwhile in practice to implement the optimization of our algorithms in this framework.

6.2.7 Implementation

The algorithm that we presented in this section, except the parametric search, has been implemented by Lingeshwaran Palanappian in C, using the gnu multiple precision library [43], with a graphical user interface using OpenGL. It allows to edit the graph and the curve, to solve the decision problem, to perform binary search on ε, and it visualizes the computed feasible parameterizations in a walk-through animation. The figure on the right shows a screen shot of an example input; the found path in the graph is highlighted. The decision algorithm runs remarkably fast without specific optimizations. For example, for graphs with $q = 700$ edges and a curve of length $q = 420$ it runs in 5 seconds, for $q = 1170$ and $p = 1000$ in 35 seconds, and for $q = 1170$ and $p = 100$ in less than 2 seconds, on a Pentium 4 processor.



6.3 Variants

We will discuss three variants of the problem, two of which can be easily integrated into the above algorithm, and another that needs more effort to be established.

6.3.1 Start and End in the Middle of Segments

One straight-forward variation of the problem is to allow a path π in G to start and end not only at vertices of G but also in the middle of segments $s_{i,j}$ for edges $(i, j) \in E$. In fact this can be easily integrated into our algorithm by replacing the condition to begin a path in the free space surface in a white $\mathbf{L}_i$ by the condition to begin a path on any white point on the left boundary of any $FD_{i,j}$ for $(i, j) \in E$. Similarly the condition to end a path in a white $\mathbf{R}_i$ is replaced by the condition to end a path on any white point on the right boundary of any $FD_{i,j}$ for $(i, j) \in E$. In order to be able to find an endpoint on the right boundary of a $FD_{i,j}$ we need to modify the definition of the right pointers and thus also of the consecutive chains. For this we do not consider only the upper boundary of $FD_{i,j}$ anymore, but imagine the upper and the right boundary to be concatenated, and let the right pointer of an interval be the lowermost reachable point on the right boundary (if it exists), and otherwise the rightmost reachable point on the upper boundary as before. Formally, for an edge $(i, j) \in E$ and a white interval I of FD_i we thus redefine $r_{i,j}(I)$ to be the rightmost/lowestmost point on $FD_j \cup F_\varepsilon(\alpha(p), s_{i,j})$. Similarly, consecutive chains can now stretch upon the right boundaries of the free space intervals. Clearly this does not affect the runtime or space bounds.

6.3.2 Avoiding U-Turns

Another variant is to ask for more monotonicity in the path π that is found in the graph. In our current problem setting we allow a path π in G to travel the same edges in G multiple times. It seems to be hard to avoid these cases without increasing the runtime immensely. However we can modify our algorithm to avoid "U-turns", i.e., to forbid a path π in G to travel the edge (i, j) and immediately afterwards the edge (j, i). Note that by definition of the Fréchet distance our algorithm implicitly forbids U-turns in the middle of an edge (since this would correspond to a non-monotone reparameterization).

We incorporate this feature in the following way: At every reachable white interval I on FD_i we do not store only one path pointer, but a pointer to each reachable interval J on FD_j, for $j \in \text{Adj}(i)$, from which I can be reached. This way we store $O(p|\text{Adj}(i)|)$ different intervals and path pointers on FD_i for each $i \in V$, which still sums up to $O(pq)$ in total. Apart from this storing scheme the preprocessing and the dynamic programming stage of the algorithm are not affected. In the path reconstruction stage we now have choices when following the path pointers. Therefore we perform a depth first traversal which runs in time linear in the number of path pointers, of which there are $O(pq)$. During this traversal we locally exclude the option to travel back the edge from which we arrived in a vertex. Altogether we obtain the same results as in Theorem 11 and Theorem 12 for the setting which excludes U-turns.

6.3.3 Time-Space-Trade-Off

Dynamic Programming

In every step of the dynamic programming stage in Section 6.2.3 we need mostly local reachability information concerning the current interval, such as its l-pointer, its r-pointer, the closest shortcut pointer, and the next white interval to the right in its consecutive chain. We can generate this information on the fly by performing the computation of the former preprocessing in an incremental way during the algorithm. In this subsection we skip the preprocessing completely, and present a variant of the dynamic programming algorithm of Section 6.2.3 that incorporates a time-space trade-off.

We store and maintain the following items during the algorithm:

- As in Section 6.2.3 we store at each vertex $i \in V$ exactly one consecutive chain C_i which is represented by its endpoints.
- In order to compute the d'_w efficiently (see proof of Lemma 28) we store for each vertex $i \in V$ a set of shortcut pointers, which we will describe in more detail below.
- For each edge $(i, j) \in E$ we maintain a stack $\mathcal{S}'(i, j)$ of indices of lower spikes, which we will describe in more detail below.
- For each edge $(i, j) \in E$ we store a current l-pointer $l_{i,j}$ and a current r-pointer $r_{i,j}$. These are the pointers with respect to $FD_{i,j}$, $j \in \text{Adj}(i)$, that have been computed for the last processed interval on FD_i. We update those pointers with every new interval that we process on FD_i.

Essentially we move the preprocessing which has been accomplished in Lemma 28 into the algorithm. I.e., we integrate the computation of the l-pointers and r-pointers into the algorithm, such that we compute those pointers only when we need to access them. If we did this in a straightforward way, we would maintain at each edge its partial maxima stack and at each vertex all shortcut pointers (compare Lemma 28), which is all information we need to construct the l-pointers and r-pointers on the fly. This however would result in a storage of $O(pq)$ altogether. In order to decrease the storage we still follow this approach but do not store the full lower maxima stacks and all shortcut pointers, but we store only equidistant samples of each. Since during the algorithm we need to recompute the missing information between two sample points, the *spacing* of this sampling is then reflected in the runtime. We will first use a spacing of $\sqrt{p}$, and will later generalize it for an arbitrary parameter $1 \leq t \leq \sqrt{p}$.

Let us now go into the details of this approach. The processing of intervals from Q is adapted as follows: For step 2 of the dynamic programming stage we need to find the leftmost white interval in C_i which lies to the right of the current interval I. For this we scan the one-dimensional cells to the right of I and directly compute each interval partition until we find the first white interval.

It remains to show how we adjust step 3 of the dynamic programming stage, since in this stage the l-pointers and r-pointers are needed. For this we follow the lines of the proof of Lemma 28. We have to show how we maintain the current l-pointers and r-pointers efficiently. For this we store and maintain compressed versions of the partial maxima stack at each edge $(i,j) \in E$, and of the shortcut pointers at each vertex $i \in V$.

For each $(i,j) \in E$ we use the notion of the partial maxima stack $\mathcal{S}(i,j)$, however we do not store $\mathcal{S}(i,j)$ directly, but only a subset of $O(\sqrt{p})$ indices. Let the stack $\mathcal{S}'(i,j)$ contain this subset of indices. $\mathcal{S}(i,j)$ is defined as in Lemma 28 to be the sequence of indices of the partial maxima of the sequence of lower spikes between two indices k and k'. We let k be the right endpoint of the last interval processed on FD_i, and k' as in Lemma 28 be the largest $k' > k$ for which (6.1) holds. In the beginning $\mathcal{S}'(i,j)$ is initialized to be empty. After that we directly compute it or update it from the previously stored stack, and we then extract the current $r_{i,j}$ from it. However, $\mathcal{S}(i,j)$ could contain up to $O(p)$ indices, which we cannot afford to store. Thus we define $\mathcal{S}'(i,j)$ to store every $\lfloor\sqrt{p}\rfloor$-th index of $\mathcal{S}(i,j)$. More precisely, $\mathcal{S}'(i,j)$ contains the first (i.e., bottommost) index of $\mathcal{S}(i,j)$, and additionally every $\lfloor\sqrt{p}\rfloor$-th index, and finally the last index of $\mathcal{S}(i,j)$, in the same order as in $\mathcal{S}(i,j)$.

In order to obtain all d'_w efficiently during the run of the algorithm we store only $O(\sqrt{p})$ *shortcut pointers* for each FD_i (as opposed to $O(p)$ pointers as in Lemma 28). For every integer $1 \leq k \leq \sqrt{p}$ we store at each position $\lfloor k\sqrt{p}\rfloor$ (which corresponds to the left boundary of the $\lfloor k\sqrt{p}\rfloor$-th cell of FD_i) a pointer to the rightmost white point on FD_i which lies to the left of $\lfloor k\sqrt{p}\rfloor$. If there is no such white point we set the shortcut pointer to NIL. We build up this pointer structure on the fly by computing a pointer value from the next shortcut pointer to its left.

In the following we show that we can process the next interval I from Q in $O(\sqrt{p})$ time.

Lemma 31 *Let x be the current position of the sweep line, and let $I \in C_i$ be the next interval in Q. Then all $\mathcal{S}'(i,j)$, $r_{i,j}$, and $l_{i,j}$ can be updated altogether in $O(\sqrt{p})$ time to comply with the new position $l(I)$ of the sweep line.*

Proof: In the beginning of the algorithm all $l_{i,j}$ and $r_{i,j}$ are initialized with -1. For an interval I that has been picked from Q we update those pointers as follows: Assume $I \in C_i$ and $j \in \mathrm{Adj}(i)$. If $l_{i,j} \geq l(I)$ then it remains unchanged. This is because it has been the leftmost reachable point of the previous interval, which due to the simplicity of $FD_{i,j}$, see

Lemma 25, implies that it is also reachable from the current interval and cannot lie further to the left. If however $l_{i,j} < l(I)$, then $l_{i,j}$ cannot be reached by a feasible monotone increasing path from I anymore. Thus in this case we greedily scan the cells of $FD_{i,j}$ to the right of $l(I)$ just as in the proof of Lemma 28 until we find the new $l_{i,j}$. The only difference is that we compute the free space in each cell on the fly. Note that, once we have computed the pointers, we free the storage required for the free space.

Again it is more challenging to update the $r_{i,j}$: Note that by construction holds that $a_{l'} \leq b_{k'}$ and $a_{l'} > b_{k'+1}$ for $l' = \text{bottom}(\mathcal{S}'(i,j))$ and $k' = \text{top}(\mathcal{S}'(i,j))$. First let $r(I) \leq k'$. We locate $r(I)$ in $\mathcal{S}'(i,j)$. If $r(I) \leq l'$ then $r_{i,j}$ remains the same. Otherwise we remove all entries from the bottom of $\mathcal{S}'(i,j)$ that are smaller than $r(I)$. Now, in order to maintain the property that $\text{bottom}(\mathcal{S}'(i,j)) = \text{bottom}(\mathcal{S}(i,j))$, we find that k with $r(I) \leq k \leq \text{bottom}(\mathcal{S}'(i,j))$ which maximizes a_k. We append k to the bottom of $\mathcal{S}'(i,j)$.

By definition of $\text{top}(\mathcal{S}'(i,j))$ we know that the largest $k' \geq k$ for which (6.1) holds has to be greater or equal to $\text{top}(\mathcal{S}'(i,j))$. We greedily search for this new value of k' exactly as in Lemma 28 and construct, on the fly, the full partial maxima stack starting at $\text{top}(\mathcal{S}'(i,j))$ and ending in k'. We then pop $\text{top}(\mathcal{S}'(i,j))$ and push the spikes of this new stack at spacing $\sqrt{p}$ onto $\mathcal{S}'(i,j)$, taking care that at the transition between the two stacks the spacing is correct, and make sure to push k' onto $\mathcal{S}'(i,j)$. We set $r_{i,j}$ to be the first d'_w which bounds a white interval on FD_j to the left of $k'+1$. We find this d'_w by greedily searching for the next white point on FD_j to the left of $k'+1$, following shortcut pointers when we meet them. Now consider the special case that the value of k' remains the same. If $l(I) \leq r_{i,j}$, then $r_{i,j}$ remains the same. Otherwise there is no point on FD_j which can be reached by a monotone feasible path from I and thus I is a black interval.

If $r(I) > k'$, then we skip $\mathcal{S}'(i,j)$ completely. We directly construct the full partial maxima stack starting at $r(I)$ and ending in k', and store the spikes at $\sqrt{p}$-spacing in $\mathcal{S}'(i,j)$ as before.

Note that the size of each $\mathcal{S}'(i,j)$ is only $O(\sqrt{p})$ during the whole course of the algorithm. Also the number of shortcut pointers stored per vertex $i \in V$ is $O(\sqrt{p})$. Thus the total storage is indeed at most $O(q\sqrt{p})$. For the analysis of the runtime consider a fixed $(i,j) \in E$. During the whole course of the algorithm $\text{bottom}(\mathcal{S}'(i,j))$ increases monotonically, and every integer between 1 and $p-1$ is at most a constant time touched, and at most once inserted in or removed from $\mathcal{S}'(i,j)$. The argument is similar to the proof of Lemma 28. Thus all changes of $\mathcal{S}'(i,j)$ take $O(p)$ time in total. However the steps of locating $r(I)$ in $\mathcal{S}'(i,j)$ and finding d'_w take $O(\sqrt{p})$ time per white interval in FD_i. □

From Lemma 31 we know that all data structures can be updated in $O(\sqrt{p})$ time for one processed interval of Q. Thus the processing of all intervals $O(qp\sqrt{p})$ in total. The computation of all shortcut pointers takes $O(p)$ time. The handling of insertions, deletions, and changes of intervals in Q takes $O(qp \log q)$ as before. Hence we obtained the following theorem.

Theorem 13 *There is an algorithm that decides if there is a path π in G such that $\delta_{\mathrm{F}}(\alpha, \beta_\pi) \leq \varepsilon$, thus solving Problem 10, in $O(pq(\sqrt{p} + \log q))$ time and $O(q\sqrt{p})$ space.*

Now let $1 \leq t \leq \sqrt{p}$ be a given trade-off parameter. We space the spikes in $\mathcal{S}'(i,j)$ at distance t instead of $\sqrt{p}$. Similarly we store shortcut pointers at each cell boundary $\lfloor kt \rfloor$ instead of $\lfloor k\sqrt{p} \rfloor$ for every integer $1 \leq k \leq p/t$. This way the storage becomes $O(qp/t)$, and the runtime is $O(pq(t + \log q))$ since in both cases it is linear in the spacing on the spikes or the shortcut pointers, respectively.

Corollary 11 *For any $1 \leq t \leq \sqrt{p}$ there is an algorithm that decides if there is a path π in G such that $\delta_{\mathrm{F}}(\alpha, \beta_\pi) \leq \varepsilon$, thus solving Problem 10, in $O(pq(t + \log q))$ time and $O(qp/t)$ space.*

Path Reconstruction

Above we only handled the decision problem Problem 10, without any attached path pointers to support the path reconstruction to solve Problem 9. However we clearly do not want to store all $O(pq)$ path pointers. We overcome this problem by applying a standard dynamic programming trick for saving space, see [52, 50]. However we will not be able to exploit it to its full extent, such that it will introduce a logarithmic factor in the runtime. We break α up into several smaller pieces and compute the solution for those subparts of α while keeping certain path pointer information for these subparts.

For $i, j \in \{0, 1, ..., p\}$ with $i \leq j$ let $\alpha[i, j] := \alpha|_{[i,j]}$ be the polygonal sub-curve of α starting in the i-th and ending in the j-th vertex of α. We start with applying the above algorithm to the whole curve $\alpha = \alpha[0, p]$.

Lemma 32 *Let $j \in V$. Then in each step of the algorithm, C_j contains at most one consecutive subchain of intervals that can be reached by a monotone feasible path in $FD_{i,j}$ from points on FD_i, for each $i \in Adj(j)$. Each consecutive subchain of C_j equals $[l_{i,j}(I), r_{i,j}(I)] \cap FD_j$ for some white interval I on FD_i.*

Proof: Assume that there are two disjoint consecutive subchains C and C' of C_j, that can be reached by a monotone feasible path in $FD_{i,j}$ from two disjoint intervals I and I', respectively, on FD_i. Let C lie to the left of C', and I lie to the left of I'. Since the left endpoints of processed intervals of Q always lie to the left of the consecutive chains, we know that $l(I) \leq l(C_j) \leq l(C)$ and also $l(I') \leq l(C_j) \leq l(C)$. But from Lemma 25 then follows that C can be reached by a monotone feasible path in $FD_{i,j}$ from I', and thus C and C' are not disjoint. If I' lies to the left of I then every feasible monotone path from I to C crosses every feasible monotone path from I' to C', thus C and C' are also not disjoint.

For the second part, let C be a consecutive subchain of C_j and assume that $[l_{i,j}(I), r_{i,j}(I)] \cap FD_j, [l_{i,j}(I'), r_{i,j}(I')] \cap FD_j \subseteq C$ with $[l_{i,j}(I), r_{i,j}(I)] \cap [l_{i,j}(I'), r_{i,j}(I')] = \emptyset$, for two disjoint intervals I, I' on FD_i. Let I lie to the left of I'. Then $l(I), l(I') \leq l(C)$, such that by Lemma 25 every feasible monotone path from I to C crosses every feasible monotone path from I' to C, such that $l_{i,j}(I) = l_{i,j}(I')$ and $r_{i,j}(I) = r_{i,j}(I')$. □

We maintain a variant of the path pointers that we had in step 4 of the algorithm in Section 6.2: For each $j \in V$ we maintain a partition of C_j into consecutive subchains that can be reached by a monotone feasible path in $FD_{i,j}$ from intervals on FD_i for $i \in \mathrm{Adj}(j)$. From Lemma 32 we know that there is one interval on FD_i from which the corresponding consecutive subchain on FD_j can be reached. Thus we can associate to each consecutive subchain exactly one feasible monotone path in the free space surface to some $\mathbf{L}_k$. In fact, for each consecutive subchain we maintain a *direct pointer* that points directly to the point $\mathbf{L}_k$ that can be reached from points on this consecutive subchain by a feasible monotone path in a concatenation of free space diagrams of the free space surface. These pointers can be maintained by constructing the path pointers as in Section 6.2, but instead of storing them, following them to the pointers of the consecutive subchain they can be reached from, and then only storing those direct pointers.

In order to be able to reconstruct one actual feasible path from the direct pointer information, we compute different direct pointers for different parts of the free space surface. For an edge $(i,j) \in E$, let $\mu_{i,j}$ be the number of the cell in $FD_{i,j}$ which contains the right endpoint of the current C_j. Note that $\mu_{i,j}$ changes during the course of the algorithm. Let $V^{i,j}_{\mu_{i,j}} := FD_\varepsilon(\alpha(\mu_{i,j}+1), s_{i,j})$ be the vertical right boundary of the partial free space diagram $FD'_{i,j} := FD_\varepsilon(\alpha[0,\mu_{i,j}+1], s_{i,j})$. Note that $V^{i,j}_{\mu_{i,j}}$ contains at most one white interval.

Note that in the regular algorithm we consider one-dimensional free space diagrams only at the upper and lower boundaries of $FD_{i,j}$ for $(i,j) \in E$. However similar to Section 6.3.1 we now have to construct one-dimensional sub free space diagrams at certain vertical cell boundaries of $FD_{i,j}$. We wish to compute for each white interval on a $V^{i,j}_{\lfloor p/2 \rfloor}$ a direct pointer to a $\mathbf{L}_k$ that can be reached by a monotone path from this interval. During the algorithm, once we arrived at $\mu_{i,j} \geq \lfloor p/2 \rfloor$, the stored partial maxima stack provides the information which interval can be reached from the white interval (if it exists at all) on $V^{i,j}_{\lfloor p/2 \rfloor}$, which in turn yields the direct pointer we want to store.

Furthermore we wish to compute for each white $\mathbf{R}_l$ a direct pointer to a white interval on a $V^{i,j}_{\lfloor p/2 \rfloor}$. For this we maintain for each consecutive subchain whose right endpoint is larger or equal to $\lfloor p/2 \rfloor$ a direct pointer to a white interval on a $V^{i,j}_{\lfloor p/2 \rfloor}$. Note that these direct pointers can be maintained in the same ways as the other direct pointers. Thus if a consecutive subchain lies completely to the left of $\lfloor p/2 \rfloor$ it stores a direct pointer to a $\mathbf{L}_k$, if it lies completely to the right it stores a direct pointer to a white interval on a $V^{i,j}_{\lfloor p/2 \rfloor}$, and if it contains $\lfloor p/2 \rfloor$ it stores both pointers. This needs $O(pq(t+\log q))$ time and $O(qp/t)$ storage for the dynamic programming. Since every consecutive chain C_j contains at most $|\mathrm{Adj}(j)|$ subchains due to Lemma 32, all direct pointers can be maintained during the dynamic programming with $O(q)$ extra space.

Concatenating the direct pointer information of both subproblems we can identify at most $O(q)$ paths that start at some $\mathbf{L}_k$, end at some $\mathbf{R}_l$, and pass a white interval on a $V^{i,j}_{\lfloor p/2 \rfloor}$ at a known point each. Note that the only information we have for these paths are their starting point, the point where they pass the white interval on $V^{i,j}_{\lfloor p/2 \rfloor}$ in the free space diagram $FD_{i,j}$, and their endpoint. We only consider exactly one of these paths, and store its starting point $\mathbf{L}_{k^*}$, its endpoint $\mathbf{R}_{l^*}$, and the indices i^*, j^* and the point a^*, where FD_{i^*,j^*} is the free space diagram where the path crosses the white interval on $V^{i^*,j^*}_{\lfloor p/2 \rfloor}$ in the point a^*.

In a recursive manner we now solve the subproblem in a second level for $\alpha[0, \lfloor p/2 \rfloor]$, maintaining direct pointers as above with respect to $\lfloor p/4 \rfloor$, and with the only start vertex k^* and the end point a^*. Note that this requires a very slight modification of the algorithm in that the endpoint is now not in a vertex of the graph, but on a fixed point on the edge (i^*, j^*), which is similar to the first variant of the algorithm discussed in Section 6.3.1. Similarly we solve the subproblem for $\alpha[\lfloor p/2 \rfloor, p]$, with respect to $\lfloor 3p/4 \rfloor$, and with the start point a^* and the end vertex l^*. Concatenating the direct pointers for both subproblems we can extract four pointers representing one feasible monotone path in the free space surface. This can be performed in $O(pq(t+\log q))$ time, $O(qp/t)$ storage, and $O(q)$ extra storage for the new pointers. We keep repeating this recursive process for $\log p$ levels until we end at single segments of α. We keep concatenating the computed pointers, and obtain a desired feasible path from some $\mathbf{L}_k$ to some $\mathbf{R}_l$ in the end. The whole recursive procedure needs $O(pq(t+\log q)\log p)$ time, $O(qp/t)$ storage, and $O(q)$ extra storage for the path representation.

Altogether we thus obtain the following theorem:

Theorem 14 *For any $1 \leq t \leq \sqrt{p}$ there is an algorithm that decides if there is a path π in G such that $\delta_{\mathrm{F}}(\alpha, \beta_\pi) \leq \varepsilon$, thus solving Problem 10, in $O(pq(t + \log q))$ time and $O(qp/t)$ space.*

If such a path π exists it can be computed together with a feasible monotone path in the free space surface, thus solving Problem 9, in $O(pq(t + \log q) \log p)$ time and $O(qp/t)$ space. For $t = 1$ the runtime is $O(pq \log q)$.

Parametric Search

We can also incorporate the time-space trade-off to solve Problem 8. We can apply parametric search in the same way as before to the algorithm which solves Problem 10 using the time-space trade-off variant. We simply plug that algorithm into the parametric search paradigm and arrive, using the same argumentation as in Section 6.2.6, at a runtime of $O(pq(t + \log q) \log(pq))$ and space complexity $O(pq/t)$. Now in order to actually find the path we first run this variant of the parametric search, which determines the optimal ε^* for which there exists a path π in G such that $\delta_{\mathrm{F}}(\alpha, \beta_\pi) \leq \varepsilon^*$. With this value for ε we run the algorithm that computes the path in $O(pq(t + \log q) \log p)$ time and $O(qp/t)$ space. Thus we can actually compute the optimal path in G in $O(pq(t + \log q) \log(pq))$ time and $O(qp/t)$ space.

Theorem 15 *For any $1 \leq t \leq \sqrt{p}$ there is an algorithm which computes a path π in G which minimizes $\delta_{\mathrm{F}}(\alpha, \beta_\pi)$, thus solving Problem 8, in $O(pq(t + \log q) \log(pq))$ runtime and $O(pq/t)$ space.*

Note that the time-space trade-off from this section together with the approach of Section 6.3.2 to avoid U-turns can be used to compute the Fréchet distance for two polygonal curves with the same time-space trade-off. Thus, at the cost of a logarithmic factor in q compared to the algorithm of [10], our algorithms also yields a time-space trade-off for computing the Fréchet distance of curves.