Matching Planar Maps*

Helmut Alt†, Alon Efrat‡, Günter Rote†, and Carola Wenk‡

Abstract

The subject of this paper are algorithms for measuring the similarity of patterns of line segments in the plane, a standard problem in, e.g. computer vision, geographic information systems, etc. More precisely, we define feasible distance measures that reflect how close a given pattern H is to some part of a larger pattern G. These distance measures are generalizations of the well known Fréchet distance for curves. We first give an efficient algorithm for the case that H is a polygonal curve and G is a geometric graph. Then, slightly relaxing the definition of distance measure we give an algorithm for the general case where both, H and G, are geometric graphs.

1 Introduction

Patterns consisting of line segments occur in many applications of a geometric nature, like computer vision, geographic information systems, CAGD, etc. In many cases the problem occurs to determine whether some given pattern H is equal to or similar to some part of a larger pattern G. Here, for the case of patterns consisting of straight line segments, we will give feasible distance measures reflecting this similarity and being compatible to paths on the pattern. Also, we will give efficient algorithms for computing these distances.

As a first task we consider a given polygonal curve, and an 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. The problem in this form already has many applications. The following one, for example, looked particularly appealing to us: 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 roadmap, and a person travelling on some of the roads, while recording its positioning information using a GPS receiver. The roadmap can be modelled by a planar embedded graph, and the path the person travelled 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 roadmap. The task is to identify those roads which have actually been travelled. This is a prerequisite for incrementally constructing roadmaps from such GPS curves, which is especially interesting for roads such as hiking trails in a forest which are not visible on aerial pictures. We present an algorithm solving this problem in Section 2. It has been implemented, and even without specific optimizations it runs surprisingly fast. In Section 3 we consider the case of two geometric graphs.

Our distance measures are based on the Fréchet distance for curves which has been investigated before in [2].

---

*Preliminary versions of this paper have appeared in [1] and as a part of [6].

†Freie Universität Berlin, Institut für Informatik, Takustraße 9, 14195 Berlin, Germany. {alt, rote}@inf.fu-berlin.de

‡University of Arizona, Computer Science Department, 1040 E 4th Street, Tucson, AZ 85721-0077. {alon, carolaw}@cs.arizona.edu

Definition 1 (Fréchet distance) *Let $f\colon I = [l_I, r_I] \to \mathbb{R}^2$, $g\colon J = [l_J, r_J] \to \mathbb{R}^2$ be two planar curves, and let $\|\cdot\|$ denote the Euclidean norm. Then the* Fréchet distance $\delta_{\mathrm{F}}(f,g)$ *is defined as*

$$\delta_{\mathrm{F}}(f,g) := \inf_{\substack{\alpha\colon [0,1]\to I \\ \beta\colon [0,1]\to J}} \max_{t\in[0,1]} \|f(\alpha(t)) - g(\beta(t))\|.$$

where α and β range over continuous and non-decreasing reparametrizations with $\alpha(0) = l_I$, $\alpha(1) = r_I$, $\beta(0) = l_J$, $\beta(1) = r_J$.

If we drop the requirement on α and β to be non-decreasing, we obtain a distance measure that is called the weak Fréchet distance *between f and g.*

A popular illustration of the Fréchet distance is the following: Suppose a person is walking a dog, the person is walking on the one curve and the dog on the other, and the person is holding the dog at a leash. Both are allowed to control their speeds but they are not allowed to go backwards. Then the Fréchet distance of the curves is the minimal length of a leash that is necessary for both to walk the curves from beginning to end.

2 Matching a Curve in a Graph

Let $G = (V,E)$ be an undirected connected planar graph with a given straight-line embedding in $\mathbb{R}^2$, $|V| = q$, $|E| = O(q)$, such that $V = \{1,\ldots,q\}$ corresponds to points $\{v_1,\ldots,v_q\} \subseteq \mathbb{R}^2$. 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 $\alpha\colon [0,p] \to \mathbb{R}^2$ be a polygonal curve in $\mathbb{R}^2$, which consists of p line segments $\overline{\alpha}_i := \alpha|_{[i,i+1]}$ for $i \in \{0,1,\ldots,p-1\}$. We consider each line segment $\overline{\alpha}_i$ to be parameterized by its *natural parametrization*, 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. We identify a path π in G with the polygonal curve that is formed by its edges. Given α and G we wish to find a path π in G which minimizes $\delta_{\mathrm{F}}(\alpha,\pi)$. Note that this definition allows a path π in G to travel the same edges in G multiple times.

We attack this minimization problem 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. Afterwards we apply parametric search, in a manner similar to that of [2], to finally solve the minimization problem. As a subproblem we consider the task of only deciding whether there exists a path in G with the desired properties. The algorithm for the decision problem then can be used to design one for the computation of such a path.

2.1 Basic Concepts and Algorithm Outline

If not stated otherwise let $\varepsilon > 0$ be given. We employ the notion of the *free space* F_ε and the free space diagram FD_ε of two curves, which was introduced in [2]:

Definition 2 ([2]) *Let $f\colon I \to \mathbb{R}^2$, $g\colon J \to \mathbb{R}^2$ be two curves; $I, J \subseteq \mathbb{R}$. The set $F_\varepsilon(f,g) := \{\,(s,t) \in I \times J \mid \|f(s) - g(t)\| \le \varepsilon\,\}$ denotes the* free space *of f and g. We call the partition of $I \times J$ into regions belonging or not belonging to $F_\varepsilon(f,g)$ the* free space diagram *$FD_\varepsilon(f,g)$.*

We call points in F_ε *white* or *feasible* and points in $\mathrm{FD}_\varepsilon \setminus \mathrm{F}_\varepsilon$ *black* or *infeasible*. See Figure 1 for an illustration.

In [2] it has been shown that $\delta_{\mathrm{F}}(f,g) \le \varepsilon$ if and only if there exists a curve within $\mathrm{F}_\varepsilon(f,g)$ from the lower left corner to the upper right corner, which is monotone in both coordinates.

We call a curve within $\mathrm{F}_\varepsilon(f,g)$ *feasible*. We thus concentrate on finding a monotone feasible path in certain free space diagrams. Figure 1 shows polygonal curves f, g, a distance ε, and the corresponding free space diagram with the free space F_ε. Observe that the monotone curve in $\mathrm{F}_\varepsilon(f,g)$ from the lower left corner to the upper right corner as a continuous mapping from $[0,1]$ to $I \times J$ directly gives continuous increasing reparametrizations α and β.

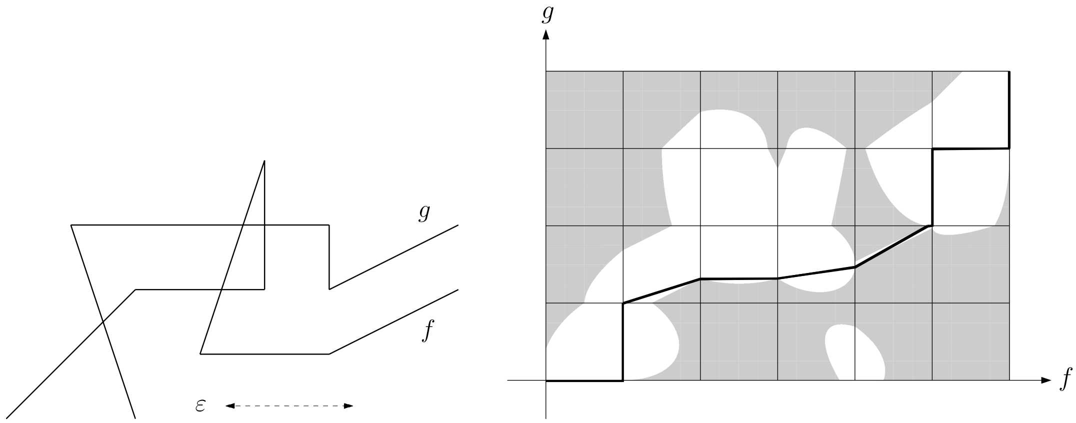

Figure 1: Free space diagram for two polygonal curves f and g. A monotone curve from the lower left corner to the upper right corner is drawn in the free space. This illustration is taken from [2].

For all $(i,j) \in E$ let $s_{i,j}$ be continuously parameterized by values in $[0,1]$ according to its natural parametrization, thus $s_{i,j} \colon [0,1] \to \mathbb{R}^2$. For every edge $(i,j) \in E$ consider the free space $\mathrm{F}_{i,j} := \mathrm{F}_\varepsilon(\alpha, s_{i,j}) \subseteq [0,p] \times [0,1]$. The free space diagram $\mathrm{FD}_{i,j} := \mathrm{FD}_\varepsilon(\alpha, s_{i,j})$ is the subdivision of $[0,p] \times [0,1]$ into the *white* points of $\mathrm{F}_{i,j}$ and into the *black* points of $[0,p] \times [0,1] \setminus \mathrm{F}_{i,j}$. See Figure 2 for an illustration.

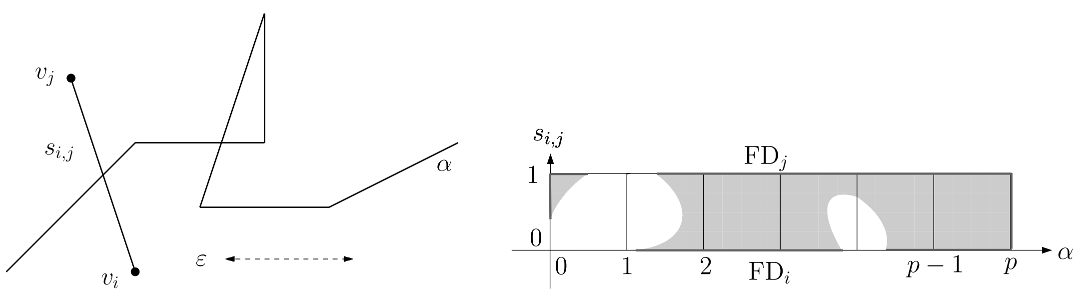

Figure 2:
Free space diagram $\mathrm{FD}_{i,j}$ for a segment $s_{i,j}$ and α.

As shown in [2], $\mathrm{FD}_{i,j}$ 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 a vertex $j \in V$ let $\mathrm{FD}_j := \mathrm{FD}_\varepsilon(\alpha, v_j)$, which is a one-dimensional free space diagram consisting of at most $2p+1$ black or white intervals. Let $\mathrm{F}_j := \mathrm{F}_\varepsilon(\alpha, v_j)$ be the corresponding one-dimensional free space, which consists of a collection of white intervals. Furthermore, let $\mathbf{L}_j$ be the left endpoint and $\mathbf{R}_j$ be the right endpoint of FD_j.

For each $i \in V$ the free space diagrams $\mathrm{FD}_{i,j}$ and $\mathrm{FD}_{j,i}$ for all $j \in \mathrm{Adj}(i)$ have the one-dimensional free space diagram FD_i in common—as the bottom of $\mathrm{FD}_{i,j}$ and as the top of

$\mathrm{FD}_{j,i}$. Thus we can glue 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* of G and α; see Figure 3 for an example.

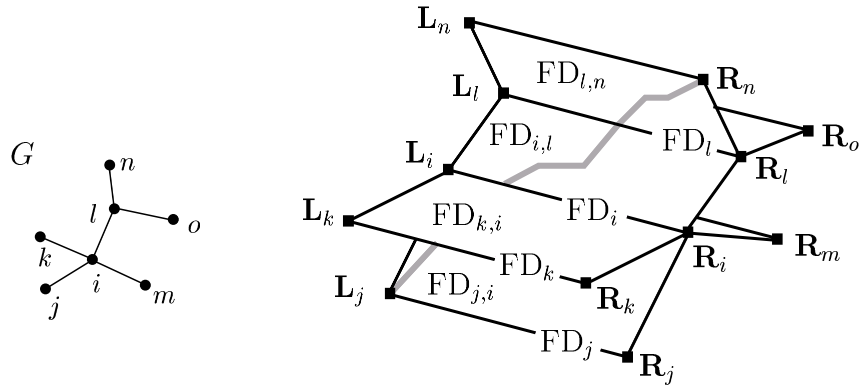

Figure 3: Example of a free space surface: Free space diagrams glued together according to the adjacency information of G. An example path π in the free space surface is highlighted in grey.

The algorithm in [2] computes a monotone feasible path in the free space diagram of two polygonal curves in a dynamic programming fashion. 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 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 $\mathrm{FD}_\varepsilon(\alpha, \pi)$. Thus let us consider the following reachability information:

For every vertex $j \in V$ let $\mathcal{R}(j)$ be the set of all points $u \in \mathrm{F}_j$ for which there exists a $k \in V$ and a path π from k to j in G such that there is a monotone feasible path from $\mathbf{L}_k$ to u in $\mathrm{F}_\varepsilon(\alpha, \pi)$. 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. We thus know that there is a path π in G with $\delta_{\mathrm{F}}(\alpha, \pi) \leq \varepsilon$ iff there is a vertex $j \in V$ such that $\mathbf{R}_j \in \mathcal{R}(j)$.

Similar to [2] we first decide whether there exists a feasible path in the free space surface 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. The algorithm solving the decision problem consists of three stages: The *preprocessing stage*, see Section 2.2, which computes the free space diagrams $\mathrm{FD}_{i,j}$ together with some additional reachability information; the *dynamic programming stage*, see Section 2.3, which decides if there exists a feasible path in the free space surface; and the *path reconstruction stage*, see Section 2.4, which constructs the path π in G along with feasible reparametrizations of π and α that witness the fact that $\delta_{\mathrm{F}}(\alpha, \pi) \leq \varepsilon$. In Section 2.6 we show how to apply parametric search to solve the minimization problem.

In the following we make use of a property of $\mathrm{FD}_{i,j}$ for each $(i, j) \in E$, which we call the *simplicity property* of $\mathrm{FD}_{i,j}$: Each $\mathrm{FD}_{i,j}$ is a row of cells, and each white region in such a cell is the intersection of an elliptical disk with the cell boundary. Thus there is no vertical line at any position in $\mathrm{FD}_{i,j}$ 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 1 *Let $(i,j) \in E$, and $u \in F_i$, $v \in F_j$ be white points with $u \leq v$ for which exists a feasible monotone path in $FD_{i,j}$ from u to v. Then for every $u' \in F_i$ and $v' \in F_j$, $u \leq u' \leq v' \leq v$ 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 $\mathrm{FD}_{i,j}$ it is possible to go straight up from u' until hitting this path, and similarly to go straight down 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 $\mathrm{FD}_{i,j}$ from u' to v'. □

2.2 Preprocessing

We compute all one-dimensional free space diagrams FD_i for all $i \in V$. Conceptually we continue to consider the $\mathrm{FD}_{i,j}$ 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 $\mathrm{FD}_{i,j} \subseteq [0,p] \times [0,1]$ consists of p cells, one for

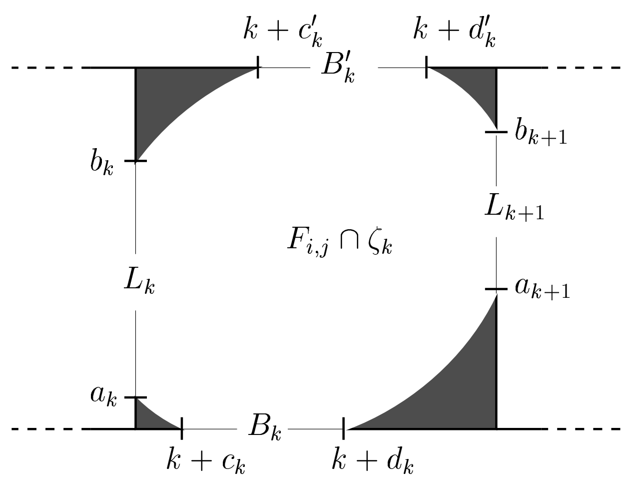

Figure 4: Intervals of the free space on the boundary of a cell.

each segment in α. Let ζ_k be the cell in $\mathrm{FD}_{i,j}$ corresponding to the kth segment $\overline{\alpha}_k$ of α, $0 \leq k \leq p-1$. Let $L_k = [a_k, b_k]$ be the white interval on the left boundary of ζ_k, let $B_k = [k + c_k, k + d_k]$ be the white interval on the bottom boundary of ζ_k, and let $B'_k = [k + c'_k, k + d'_k]$ be the white interval on the top boundary of ζ_k. See Figure 4 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 $\mathrm{FD}_{i,j}$. We call $\{k\} \times \mathbb{R}$ the *vertical line at k*. We call the black parts in ζ_k, of which there are at most four, *spikes*. In particular we call the spikes bounded from above by a_k or a_{k+1} *lower spikes*, and the spikes bounded from below by b_k or b_{k+1} *upper spikes*. We call $a_k, a_{k+1}, b_k, b_{k+1}$ the *heights* of the corresponding spikes. Similarly, we call c_k, c'_k, d_k, d'_k *widths* of *left* and *right* spikes. We call k the *index* of the two spikes bounding L_k. Note that the interval endpoints correspond to heights or widths of spikes.

For each $(i,j) \in E$ we compute for each white interval I of FD_i the leftmost point $l_{i,j}(I)$ (*left pointer* or *l-pointer*) on FD_j and the rightmost point $r_{i,j}(I)$ (*right pointer* or *r-pointer*) on FD_j which can be reached from some point in I by a monotone feasible path in $\mathrm{FD}_{i,j}$.

Preprocessing:

1. For all $i \in V$ compute the one-dimensional free space diagrams FD_i .
2. For every $i \in V$ and every white interval I of FD_i compute for all $j \in \mathrm{Adj}(i)$ the pointers $l_{i,j}(I)$ and $r_{i,j}(I)$, and store them in an array each, indexed by j. See Lemma 3.

Figure 5: Preprocessing steps.

This can be done in linear time for all intervals on FD_j, see Lemma 3. 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 [2] 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 some $0 \le k \le p-1$. 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. See Figure 5 for an overview of the preprocessing steps.

The following lemma gives a characterization when points on FD_j can be reached from points on FD_i by a monotone feasible path in $\mathrm{FD}_{i,j}$.

Lemma 2 *Let $(i,j) \in E$ be fixed. Let $0 \le k < k' \le p-1$, and assume that $B_k, B'_{k'} \neq \emptyset$. Then there is a monotone feasible path in $FD_{i,j}$ from some point on B_k to a point on $B'_{k'}$ if and only if*

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

Proof: Assume there is a monotone path π in $\mathrm{F}_{i,j}$ 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 (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 under 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 $a_{i_{\nu+1}}$. Since (1) holds for $l = i_1, \ldots, i_m$, every described piece in the path is indeed feasible, and π is monotone. $\square$

Lemma 3 *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, $1 \le k \le p-1$, can be computed in $O(p)$ time.*

Proof: The left pointers $l_{i,j}(B_k)$ for all $0 \le k \le p-1$ are easily computed by a scan for increasing $k = 0, \ldots, p-1$: Let k be fixed. If $c_k \le d'_k$ 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 (1) holds. If such a cell does not exist then we set $l_{i,j}(B_k) :=$ NIL. 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 (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 $\mathcal{S} = \{i_1, i_2, \ldots, i_m\}$ we have $i_1 < i_2 < \cdots < i_m$ and $a_{i_1} > a_{i_2} > \cdots > a_{i_m}$. See Figure 6 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 $\mathrm{FD}_{i,j}$ is $a_{i_{s+1}}$.

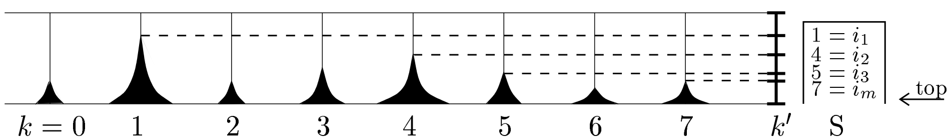

Figure 6: An example of lower spikes and their partial maxima stack $\mathcal{S}$.

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 (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 (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 (1). (We always denote the top element of $\mathcal{S}$ by a_{i_m} and the bottom element by a_{i_1}, although the indices and the value of m change during the algorithm.) If $a_{i_1} > b_{k'+1}$, then $k'+1$ violates (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 (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 (1), we know that there is no monotone feasible path in $\mathrm{FD}_{i,j}$ 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 $k'+1$.

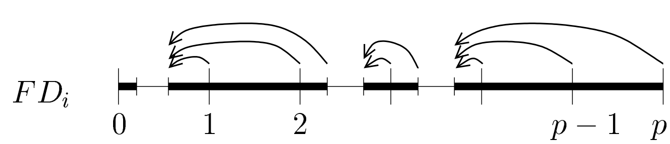

Figure 7: Shortcut pointers on FD_i.

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 k-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. See Figure 7 for an illustration. 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 beginning of FD_j. If $k < w$ then we set $r_{i,j}(B_k) := w + d'_w$. If $k > w$ then we set $r_{i,j}(B_k) :=$ NIL. If $k = w$ then if $c_k \le d'_k$ we set $r_{i,j}(B_k) := k + d'_k$, otherwise we set $r_{i,j}(B_k) :=$ NIL.

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 and 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 $O(p)$.

□

2.3 Dynamic Programming

In this stage we decide whether there exists a feasible monotone path in the free space surface. Note that such a path traverses a sequence of free space diagrams $\mathrm{FD}_{i,j}$. We call the part of a path that traverses one such free space diagram a *segment* of the path.

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

Definition 3 **(C_i)** *Let $i \in V$ and x be the current position of the sweep line. Then C_i consists of all reachable points $u \in \mathcal{R}(i) \subseteq FD_i$, such that $u \ge x$, 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 whether $\mathbf{R}_i \in \mathcal{R}(i)$ by checking if $\mathbf{R}_i \in C_i$ for an advanced enough position x of the sweep line. Let us call a sequence of consecutive white and black intervals of FD_i a *consecutive chain* of intervals. For a consecutive chain, as well as for a single interval, C let $l(C)$ be its left and $r(C)$ be its right endpoint. For two consecutive chains $C' \subseteq C$ we call C' a *consecutive subchain* of C.

Lemma 4 *Every C_i, for $i \in V$, is a consecutive chain, for every value of x.*

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 $j \in \mathrm{Adj}(i)$ and a white point $u \in \mathrm{F}_j$ with $u \le x \le w$, such that u is reachable and there exists a monotone feasible path in $\mathrm{FD}_{j,i}$ from u to w. For any white point $v \in \mathrm{F}_i$ with $x \le v \le w$ there exists by Lemma 1 a monotone feasible path from u to v in $\mathrm{FD}_{j,i}$, which makes v in particular also reachable by the same path that reaches u, concatenated with the monotone feasible path from u to v. Thus $v \in C_i$, and C_i is a consecutive chain. See Figure 8 for an illustration.

□

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

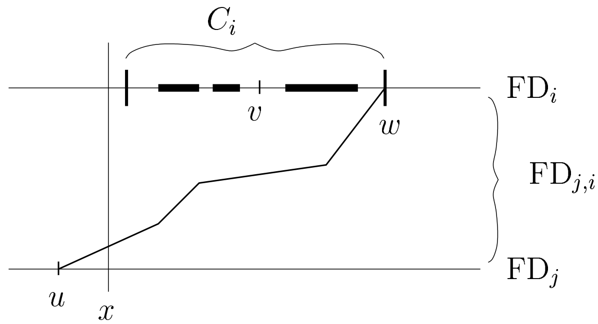

Figure 8: A consecutive chain C_i.

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 together with the interval. Since $C_i = [l(C_i), r(C_i)] \cap \mathrm{FD}_i$ 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 $j \in \mathrm{Adj}(i)$ update C_j to comply with the new value of x: $[l_{i,j}(I), r_{i,j}(I)]$ defines a consecutive chain on FD_j, whose white intervals are white intervals on FD_j which have now been identified to be reachable. Thus we need to merge $[l_{i,j}(I), r_{i,j}(I)]$ into C_j. Knowing that C_j is a consecutive chain for every value of x, we can merge both chains together by simply considering the interval endpoints. If $l_{i,j}(I) > r(C_j)$ then we discard the old C_j and replace it with $[l_{i,j}(I), r_{i,j}(I)]$. If the left endpoint has changed then we delete the old first interval of C_j in Q and insert the new one. Assuming an appropriate implementation of the priority queue, each operation on Q takes $O(\log p)$ time.

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 $\mathrm{FD}_{i,j}$).

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 π in G with $\delta_{\mathrm{F}}(\alpha, \pi) \leq \varepsilon$. In the first case we know that such a path exists, and we reconstruct it using the path pointers in the second stage of the algorithm, which is described in Section 2.4.

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 $\mathrm{FD}_\varepsilon(\alpha, \pi)$ which witnesses the fact that $\delta_{\mathrm{F}}(\alpha, \pi) \leq \varepsilon$.

By construction the interval 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 of 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. The corresponding feasible monotone path in $\mathrm{FD}_\varepsilon(\alpha, \pi)$ can be constructed in an incremental way by following the path sequence and assuring monotonicity by using again a partial maxima stack of indices of lower spikes, such as in Lemma 3.

2.5 Time Analysis

Theorem 1 *The described algorithm decides whether there is a path π in G such that $\delta_{\mathrm{F}}(\alpha, \pi) \leq \varepsilon$ in $O(pq \log q)$ time and $O(pq)$ space, where p is the number of line segments of α and q is the complexity of G. If such a path π exists the algorithm computes 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 3 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. Each priority queue 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 $O(p|E|) = O(pq)$ time, which together with the priority queue operations is $O(pq \log q)$ time for the whole dynamic programming stage. We store only one consecutive chain per vertex, and Q contains at most one interval per vertex, which adds up to $O(q)$ space. Additionally we store one path pointer per interval in FD_i, thus the space complexity for the path pointers is $O(pq)$.

By construction of the path pointers there is no cycle in the graph of path pointers. Thus every path pointer can be contained in a monotone feasible path in the free space surface at most once. We reconstruct a feasible path using a graph traversal in $O(pq)$ time (since there are $O(pq)$ path pointers). Clearly the construction of π in G then also needs $O(pq)$ time. □

The program has been implemented in C 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 parametrizations in a walk-through animation. See Figure 9 for a screenshot of an example input; the found path π in G is marked in bold. The decision algorithm runs remarkably fast without specific optimizations. For example, for graphs with $q = 700$ edges and a curve of length $p = 420$ 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. The implementation and the algorithm are shown in a video [7].

Observe that in practice one would prefer to run the algorithm on a pruned graph G' which consists of those edges of G which are in the ε-neighborhood of α. Those edges can easily be found with a line sweep on G and the ε-neighborhood of α. Notice however that this does not yield a speed-up of the asymptotic runtime.

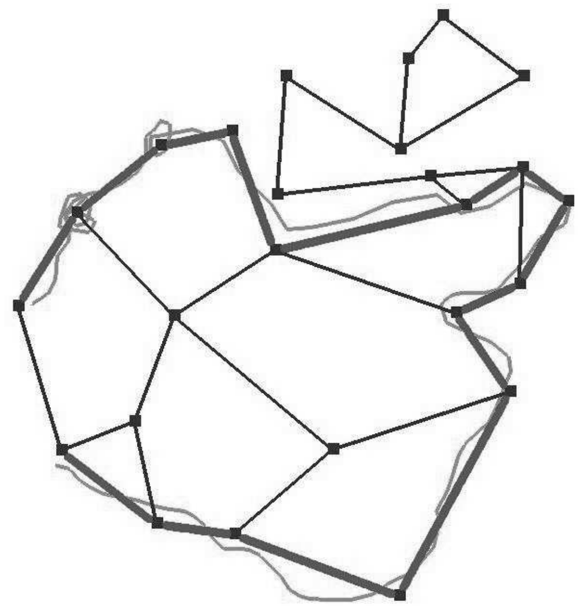

Figure 9: Screenshot of the program. The curve α is drawn in light grey, and the edges of π are marked in bold.

2.6 Parametric Search

In order to find the optimal ε we apply parametric search, analogously to [2], to the algorithm we presented to solve the decision problem. The outcome of this algorithm 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 [2] 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 [3] we obtain a running time of $O(pq \log(pq) \log q)$, at no extra storage.

Theorem 2 *There is an algorithm that finds a path π in G which minimizes $\delta_{\mathrm{F}}(\alpha, \pi)$, in $O(pq \log(pq) \log q)$ time and $O(pq)$ space.*

2.7 Variants

There are several variants of the problem setting and of the basic algorithm.

First, observe that the algorithm works in the same way for arbitrary (strongly) connected but possibly non-planar or directed graphs with straight-line embeddings, as well as for embeddings of the graph and the curve in higher-dimensional spaces. Since the algorithm to compute the Fréchet distance does not depend on the dimension of the space in which the curves are embedded, the runtime of the algorithm remains the same with q denoting the number of edges and vertices of G.

Another straight-forward variant 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 letting a path begin (or end) at any white point on the left (or right) boundary of any $\mathrm{FD}_{i,j}$.

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). We incorporate this feature by storing, at every reachable white interval I on FD_i, a path pointer to *each* reachable interval on FD_j from which I can be reached; $j \in \mathrm{Adj}(i)$. Performing a depth first traversal in this graph of path pointers we can locally exclude the option to travel back the edge from which we arrived in a vertex, and thus altogether obtain the same results as before.

Another variant is a time-space tradeoff, which we sketch in Section 2.8 and describe in detail in Appendix A.

2.8 Time-Space Tradeoff

In every step of the dynamic programming stage in Section 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 conducting the former preprocessing in an incremental way during 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 straight-forward way, we would maintain at each edge its partial maxima stack and at each vertex all shortcut pointers (compare Lemma 3), which is all information we need to construct the l-pointers and r-pointers on the fly. This however would result in a total storage of $O(pq)$. In order to decrease the storage we still follow this approach but do not store the full partial 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. In the path reconstruction stage we apply a standard dynamic programming trick for saving space, see [4, 5], which in our case introduces a logarithmic factor in the runtime. We refer the interested reader to Appendix A for the detailed description of this approach. The obtained results are summarized below.

Theorem 3 *For any $1 \leq t \leq p$ there is an algorithm that decides if there is a path π in G such that $\delta_{\mathrm{F}}(\alpha, \beta_\pi) \leq \varepsilon$ in $O(pq(t + \log q))$ time and $O(pq/t)$ space.*

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

Theorem 4 *For any $1 \leq t \leq p$ there is an algorithm which computes a path π in G which minimizes $\delta_{\mathrm{F}}(\alpha, \beta_\pi)$ in $O(pq(t + \log q) \log(pq))$ runtime and $O(pq/t)$ space.*

Note that the time-space tradeoff from this section together with the variant to avoid U-turns can be used to compute the Fréchet distance for two polygonal curves with the same time-space tradeoff. Thus, at the cost of a logarithmic factor in q compared to the algorithm of [2], our algorithms also yields a time-space tradeoff for computing the Fréchet distance of curves.

3 Graph-to-Graph Distance

In this section we generalize the Fréchet distance to pairs of *geometric graphs*, i.e., embedded, connected graphs $H = (V_H, E_H)$ and $G = (V_G, E_G)$ with straight edges. Observe, that if H is not a curve there is, in general, no injective continuous parameterization $f\colon [0,1] \to H$, so that we have to relax this condition. In the person-dog paradigm we would like to define the distance from H to G as the shortest length of a leash necessary so that the dog visits each point of the edges of H while the person traverses some part of G.

More formally, identifying H and G with the points lying on their edges we will call a mapping $f\colon [0,1] \to H$ which is continuous and surjective, a *traversal* of H. A continuous (but not necessarily surjective) mapping $g\colon [0,1] \to G$ will be called a *partial traversal* of G. The *traversal distance* from H to G is defined as

$$\delta_T(H,G) = \inf_{f,g} \max_{t\in[0,1]} \|f(t) - g(t)\|$$

where f ranges over all traversals of H and g over all partial traversals of G. Observe, that if H and G are polygonal chains this definition corresponds to the weak Fréchet-distance, see [2]. Also observe that the traversal distance is not a generalization of the Fréchet distance between a curve and a graph as defined in Section 2. Figures 10a and 10d show examples, where the traversal distance from H to G is small, in Figures 10b and 10c it is large. Let

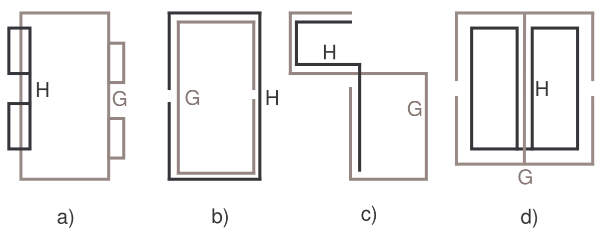

Figure 10: a), d) small traversal distance; b), c) large traversal distance.

us first consider the decision problem, i.e., determining for given H, G and $\varepsilon \geq 0$ whether $\delta_T(H,G) \leq \varepsilon$. In order to find an algorithm for the decision problem, we consider for all edges $e \in E_H$ and $f \in E_G$ the cells $C_{e,f}$ of the free space diagram, which can be identified with the two-dimensional unit interval $[0,1]^2$ within which, as was mentioned before, the freespace is obtained by the intersection with an elliptical disk. If $e = (u,v)$ and $f = (x,y)$, we name the right, left, upper and lower sides of $C_{e,f}$ as $C_{v,f}$, $C_{u,f}$, $C_{e,y}$, and $C_{e,x}$, respectively (see Figure 11). Then we identify sides with the same name, i.e. we "glue together" cells of the form $C_{e,f}$ and $C_{e,f'}$ ($C_{e,f}$ and $C_{e',f}$) if f and f' (e and e') have a common endpoint x (u) at the sides named $C_{e,x}$ ($C_{u,f}$). Thus we obtain a generalization of the free space surface from Section 2.1, which is a two-dimensional cell complex $\mathcal{S}$ in three dimensions, whose facets are the cells $C_{e,f}$, whose edges are the sides $C_{u,f}$ and $C_{e,x}$, and whose vertices are the points $C_{u,x}$, with $e \in E_H$, $f \in E_G$, $u \in V_H$, $x \in V_G$. Please note that we use a slightly different notation in this section than in Section 2.

$\mathcal{S}$ contains the combined "white" freespace of all its cells and is a generalization of the freespace diagram of two curves. A continuous path on $\mathcal{S}$ which completely lies inside the free space corresponds to a simultaneous motion on G and H keeping a distance of at most ε. Let us call these paths *feasible*.

If a feasible path π traverses some cell $C_{e,f}$ then let $I_{\pi,e,f}$ be the set of those points on e that are traversed by the corresponding motion on the graphs. The edge $e \in E_H$ is called *satisfied* by π if

$$\bigcup_{f \in E_G} I_{\pi,e,f} = e \ .$$

It means that all points of e are eventually traversed by the motion on the graphs corresponding to π. Therefore, we can conclude:

Lemma 5 *$\delta_T(H,G) \leq \varepsilon$, if and only if there exists a feasible path π satisfying all edges $e \in E_H$.*

In order to obtain an algorithm to test the condition of Lemma 5 we introduce the *traversal graph* $\mathcal{T}$. The vertices of $\mathcal{T}$ are the one-dimensional facets $C_{u,f}$ and $C_{e,x}$ of the cell complex $\mathcal{S}$, with $e \in E_H$, $f \in E_G$, $u \in V_H$, $x \in V_G$. Two such facets are connected by an edge of $\mathcal{T}$ if and only if they are both incident to some cell $C_{e,f}$ and if there is a connection between both by a curve through the free space of $C_{e,f}$, see Figure 11. Thus, to each edge of $\mathcal{T}$ we can assign a cell of the free space. On the other hand, each cell is assigned to at most six edges. It follows that $\mathcal{T}$ has $O(pq)$ edges where $p = |E_G|$ and $q = |E_H|$.

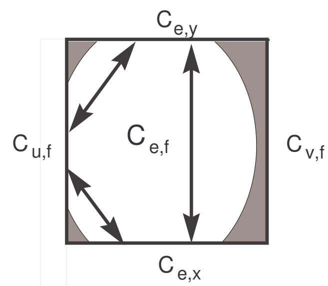

Figure 11: Edges of the traversal graph.

For $e \in E_H, f \in E_G$ let $J_{e,f}$ be the set of all points on e that have distance at most ε from f, i.e., the projection of the freespace in $C_{e,f}$ to e. Any path π with the properties described in Lemma 5 yields a path in the traversal graph $\mathcal{T}$ whose edges are assigned to the cells $C_{e,f}$ traversed by π. Since any edge $e \in E_H$ is satisfied by π it must be $\bigcup_f J_{e,f} = e$ where f ranges over all cells $C_{e,f}$ traversed by π. Consequently, the equation is true if f ranges over all edges in E_G such that $C_{e,f}$ is an edge in the connected component $\mathcal{C}$ of $\mathcal{T}$ containing π. Our algorithm for the decision problem is based on the fact that also the converse is true:

Lemma 6 *$\delta_T(H,G) \leq \varepsilon$, if and only if there exists a connected component $\mathcal{C} = (V_\mathcal{C}, E_\mathcal{C})$ of the traversal graph $\mathcal{T}$ such that for all $e \in E_H$*

$$\bigcup_f J_{e,f} = e$$

where f ranges over all edges in E_G where $C_{e,f}$ is assigned to an edge in $E_\mathcal{C}$.

To see the converse suppose that $\mathcal{C}$ is a connected component of $\mathcal{T}$ with this property. Then we construct a path π on $\mathcal{S}$ as follows: π traverses all vertices of $\mathcal{C}$ by, say, breadth-first-search. For each cell $C_{e,f}$ visited, π makes sure that $I_{\pi,e,f} = J_{e,f}$ by visiting the leftmost and the rightmost point of the freespace (see Figure 12).
Then for all $e \in E_H$

$$\bigcup_f I_{e,f,\pi} = \bigcup_f J_{e,f}$$

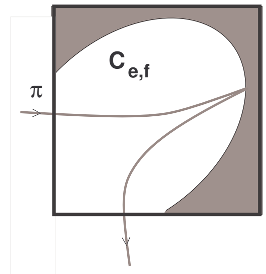

Figure 12: Motion of π within $C_{e,f}$.

where f ranges over all edges in E_G such that $C_{e,f}$ is a cell visited by π. Therefore π satisfies all edges $e \in E_H$ and $\delta_T(H, G) \leq \varepsilon$ by Lemma 5.

Lemma 6 enables us to give a quite simple **algorithm** for solving the decision problem. In fact, given geometric graphs G and H and $\varepsilon > 0$, we first determine all freespace cells $C_{e,f}, e \in E_H, f \in E_G$, and the traversal graph $\mathcal{T}$. By breadth-first-search we determine all connected components of $\mathcal{T}$ and we check for each of them whether the condition of Lemma 6 holds for each edge $e \in E_H$. If this is the case for at least one connected component, the algorithm answers "yes", otherwise "no".

In order to determine the runtime of this algorithm, we observe that the breadth-first-search in total visits $O(pq)$ cells $C_{e,f}$ since there are $O(pq)$ edges in $\mathcal{T}$. For each cell we have to add the interval $J_{e,f}$ to the portion of e covered so far which takes time $O(\log pq)$.

In order to solve the *optimization problem* observe that for the smallest ε for which the decision problem has a positive answer, there are two possibilities. On the one hand it could be that the left endpoint of some interval $J_{e,f}$ equals the right endpoint of another one $J_{e,f'}$, so that edge e gets satisfied at that point. On the other hand it could be that the free space in some cell $C_{e,f}$ touches one of the sides of the cell, i.e., the traversal graph $\mathcal{T}$ changes. Therefore, in order to solve the optimization problem we perform a parametric search using Cole's approach [3] with a fast parallel sorting algorithm for the endpoints of the intervals $J_{e,f}$, including the values 0 and 1 to take care of the critical values of the second type. Since there are $O(pq)$ such endpoints and the decision problem can be solved in time $O(pq \log pq)$ we obtain an $O(pq \log^2 pq)$ algorithm for the computation problem. We summarize:

Theorem 5 *Given two geometric graphs G and H and $\varepsilon > 0$, it can be* decided *whether $\delta_T(H, G) \leq \varepsilon$ in time $O(pq \log pq)$ by the algorithm given above, where p and q are the numbers of edges of G and H, respectively. The traversal distance from H to G can be* computed *in time $O(pq \log^2 pq)$.*

4 Acknowledgements

We thank Scott Howard Morris for introducing us to the application of matching GPS curves, and Lingeshwaran Palaniappan for implementing the algorithm of Section 2.

References

[1] H. Alt, A. Efrat, G. Rote, and C. Wenk. Matching planar maps. In *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms*, pp. 589–598, 2003.

[2] H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *Internat. J. Comput. Geom. Appl.*, 5:75–91, 1995.

[3] R. Cole. Slowing down sorting networks to obtain faster sorting algorithms. *J. Assoc. Comput. Mach.*, 34(1):200–208, 1987.

[4] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[5] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. Assoc. Comput. Mach.*, 24:664–675, 1977.

[6] C. Wenk. *Shape Matching in Higher Dimensions.* PhD thesis, Freie Universität Berlin, 2002; to appear.

[7] C. Wenk, H. Alt, A. Efrat, L. Palaniappan, and G. Rote. Finding a curve in a map (Video). To appear in *Proc. 19th Ann. Symp. Comput. Geom.*, San Diego, June 2003. Association for Computing Machinery, 2 pp.

A Appendix: Time-Space Tradeoff

This section presents a detailed description of the time-space tradeoff which was sketched in Section 2.8.

A.1 Dynamic Programming

Observe that in every step of the dynamic programming stage 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. In this section we skip the preprocessing completely, and present a variant of the dynamic programming algorithm of Section 2.3 that integrates the preprocessing into the algorithm in such a way that it incorporates a time-space tradeoff.

We store and maintain the following items during the algorithm:

- As in Section 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 3) 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 $\mathrm{FD}_{i,j}$, $j \in \mathrm{Adj}(i)$, 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.

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 3), which is all information we need to construct the l-pointers and r-pointers on the fly. This however would result in a total storage of $O(pq)$. In order to decrease the storage we still follow this approach but do not store the full partial 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 to an arbitrary parameter $1 \le t \le p$.

Let us now go into the details of this approach. The processing of intervals from the priority queue 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 3. 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 3 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 3 be the largest $k' > k$ for which (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 3). For every integer $1 \le k \le \sqrt{p}$ 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 the priority queue Q in $O(\sqrt{p})$ time.

Lemma 7 *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 to comply with the new position $l(I)$ of the sweep line in total time $O(\sqrt{p})$.*

Proof: In the beginning of the algorithm all $l_{i,j}$ and $r_{i,j}$ are initialized with NIL. 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 $l_{i,j} \ge l(I)$ then it remains unchanged. This is because it has been the leftmost reachable point of the previous interval, which due to the simplicity of $\mathrm{FD}_{i,j}$, see Lemma 1, 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 $\mathrm{FD}_{i,j}$ to the right of $l(I)$ just as in the proof of Lemma 3 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 $a_{l'} \le b_{k'}$ and $a_{l'} > b_{k'+1}$ for $l' = \mathrm{bottom}(\mathcal{S}'(i, j))$ and $k' = \mathrm{top}(\mathcal{S}'(i, j))$. First let $r(I) \le k'$. 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 (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 3 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, hence $r_{i,j} := \text{NIL}$.

If $r(I) > k'$, then we discard $\mathcal{S}'(i,j)$. We directly construct the full partial maxima stack starting at $r(I)$ and ending in k', and store the indices 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 touched at most a constant number of times, and at most once inserted in or removed from $\mathcal{S}'(i,j)$. The argument is similar to the proof of Lemma 3. 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. $\square$

From Lemma 7 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 takes $O(pq\sqrt{p})$ time in total. The computation of all shortcut pointers takes $O(p)$ time. The handling of insertions, deletions, and changes of intervals in Q takes $O(pq \log q)$ as before. Hence we obtained the following result:

Lemma 8 *There is an algorithm that decides if there is a path π in G such that $\delta_{\text{F}}(\alpha, \beta_\pi) \leq \varepsilon$ in $O(pq(\sqrt{p} + \log q))$ time and $O(q\sqrt{p})$ space.*

Now let $1 \leq t \leq p$ be a given tradeoff 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 $O(pq/t)$, and the runtime is $O(pq(t + \log q))$ since in both cases the time to process an interval in Q is linear in the spacing of the spikes and the shortcut pointers.

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

A.2 Path Reconstruction

Above we only handled the decision problem without any attached path pointers to support the path reconstruction. 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 [4, 5]. 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 $i, j \in \{0, 1, \ldots, p\}$ 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 9 *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 $\mathrm{FD}_{i,j}$ 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 1 then follows that C can be reached by a monotone feasible path in $\mathrm{FD}_{i,j}$ 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 $[l_{i,j}(I), r_{i,j}(I)] \cap \mathrm{FD}_j, [l_{i,j}(I'), r_{i,j}(I')] \cap \mathrm{FD}_j \subseteq C$ 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 1 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 $r_{i,j}(I) = r_{i,j}(I')$. $\square$

We maintain a variant of the path pointers that we had in step 4 of the algorithm in Section 2.3: 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 $\mathrm{FD}_{i,j}$ from intervals on FD_i for $i \in \mathrm{Adj}(j)$. From Lemma 9 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 2.3, but instead of storing them we follow them to the pointers of the consecutive subchain they can be reached from, and then we store 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 $\mathrm{FD}_{i,j}$ which contains the right endpoint of the current C_j. Note that $\mu_{i,j}$ changes during the course of the algorithm. Let $V^{i,j}_{\mu_{i,j}} := \mathrm{FD}_\varepsilon(\alpha(\mu_{i,j}+1), s_{i,j})$ be the vertical right boundary of the partial free space diagram $\mathrm{FD}'_{i,j} := \mathrm{FD}_\varepsilon(\alpha[0, \mu_{i,j}+1], s_{i,j})$. 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 $\mathrm{FD}_{i,j}$ for $(i,j) \in E$. However we now have to construct one-dimensional sub free space diagrams at certain vertical cell boundaries of $\mathrm{FD}_{i,j}$. We wish to compute for each white interval on a $V^{i,j}_{\lfloor p/2 \rfloor}$ a direct pointer to an $\mathbf{L}_k$ 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 way 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 $O(pq/t)$ storage for the dynamic programming. Since every consecutive chain C_j contains at most $|\mathrm{Adj}(j)|$ subchains due to Lemma 9, 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 $\mathrm{FD}_{i,j}$, 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 one of the variants discussed in Section 2.7. 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, $O(pq/t)$ 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, $O(pq/t)$ storage, and $O(q)$ extra storage for the path representation. Altogether this yields the following result:

Theorem 3 *For any $1 \leq t \leq p$ there is an algorithm that decides if there is a path π in G such that $\delta_{\mathrm{F}}(\alpha, \beta_\pi) \leq \varepsilon$ in $O(pq(t + \log q))$ time and $O(pq/t)$ space.*

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

A.3 Parametric Search

In order to find the optimal ε we can apply parametric search in the same way as before. We simply plug the time-space tradeoff variant into the parametric search paradigm and arrive, using the same argumentation as in Section 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 $O(pq/t)$ space. Thus we can actually compute the optimal path in G in $O(pq(t + \log q) \log(pq))$ time and $O(pq/t)$ space.

Theorem 4 *For any $1 \leq t \leq p$ there is an algorithm which computes a path π in G which minimizes $\delta_{\mathrm{F}}(\alpha, \beta_\pi)$ in $O(pq(t + \log q) \log(pq))$ runtime and $O(pq/t)$ space.*