Computing Fréchet Distance with Speed Limits*

A. Maheshwari, J.-R. Sack, and K. Shahbaz

Abstract

In this paper, we study a problem on computing the Fréchet distance between two polygonal curves and provide efficient solutions for solving it. In the classical Fréchet distance, point objects move arbitrarily fast on the polygonal curves. Here, we consider the problem instance where the speed per segment is a constant within a specified range. We first describe a naive algorithm which solves the decision problem in $O(n^3)$ time. Then, we develop a faster algorithm which is based on the naive one, but exhibits a running time of $O(n^2 \log n)$. Finally, we show that the exact Fréchet distance for this problem instance can be computed in $O(n^2 \log^2 n)$.

1 Introduction

Alt and Godau [1] consider the Fréchet distance to be one of the most fundamental measures for determining the similarity between two polygonal curves. The Fréchet distance between two curves is often referred to as "dog-leash distance" because it can be interpreted as the minimum length leash required for a person to walk a dog, if the person and the dog, each travel from a starting point to an ending point of his/its curve, respectively, without ever letting go of the leash or backtracking. Two problem instances naturally arise: decision and optimization. In the decision problem, one wants to decide whether two polygonal curves U and V are within ϵ Fréchet-distance from each other, i.e., if a leash of length ϵ suffices. In the optimization problem, one wants to determine the minimum such ϵ. In [1], a quadratic-time algorithm for the decision problem was proposed, where n is the number of segments on the curves. Furthermore, they solve the corresponding optimization problem in $O(n^2 \log n)$ time.

In the above problem, the speeds of motion used on the two polygonal curves is unbounded. Motivated by the practical importance of similarity measures, we here consider a problem variant in which motion speeds are bounded, both from below and from above. I.e., there is a speed range, minimum and maximum speed, assigned to each of the segments of the two polygonal curves. We say that a point object traverses a curve with permissible speed, if it traverses the polygonal curve from start to end so that the speed used on each segment falls within the permissible range.

The decision version of Fréchet problem with speed limits is formulated as follows: Let U and V be two polygonal curves with minimum and maximum permissible speeds assigned to each segment of U and V. (Formally, $U = \{U_0, U_1, ..., U_u\}$ and $V = \{V_0, V_1, ..., V_v\}$, and Min_i and Max_i are the speed limits on each segment U_i, Min'_j and Max'_j are the speed limits on each segment V_j) For a given $\epsilon \geq 0$, is there an assignment of speeds so that two point objects can traverse U and V with permissible speed and, throughout the entire traversal, are at a distance of at most ϵ from each other. This problem may have practical usage in GIS, when the speed of moving objects is considered in addition to the geometric structure of the trajectories.

In Section 2, we explain a naive algorithm to solve the decision problem. Then, in Section 3, we describe how we can improve our naive algorithm to obtain a faster running time.

2 Naive Decision Algorithm

Our algorithm is based on the construction of the *free space diagram* used for the traditional Fréchet decision problem introduced by [1], which we discuss next.

Let U and V be two polygonal curves in plane and $U(s)$ and $V(t)$, $s, t \in [0, 1]$, be affine mapping representing them. F_ϵ, is the set of the pairs of points (s,t), for which the Euclidean distance between $U(s)$ and $V(t)$ is at most ϵ, i.e., $F_\epsilon = \{(s, t) \in [0, 1]^2 \mid d(U(s), V(t)) \leq \epsilon\}$. Figure 1a depicts two polygonal paths U and V. F_ϵ, is best represented (see Figure 1b) as a "free space diagram", which is the combination of the free space cells for all pairs of segments from U and V, respectively. The colour "white" represents F_ϵ and grey indicates its complement in $[0, 1]^2$. U has been affinely mapped to the vertical axis of the free space diagram and V to the horizontal axis. One such cell is illustrated in Figure 1c. The cell values are determined to identify passages between neighboring cells. $L^F_{i,j}$ refers to the left line segment and $B^F_{i,j}$ refers to the bottom line segment that bound $F_\epsilon(i, j)$. These values are computed by finding intersection of a square and an ellipse for each cell. In [1], it was observed that any path from the start point to the target point in the free space diagram that is monotone in both directions, corresponds to traversals

*School of Computer Sciences, Carleton University, Ottawa. Research supported by NSERC and SUN Microsystems.

of U and V, respectively, which are at distance at most ϵ from each other. (Figure 1b shows such a monotone curve inside the free space diagram.) Based on this, Alt and Godau [1] derived their quadratic time algorithm for the decision problem.

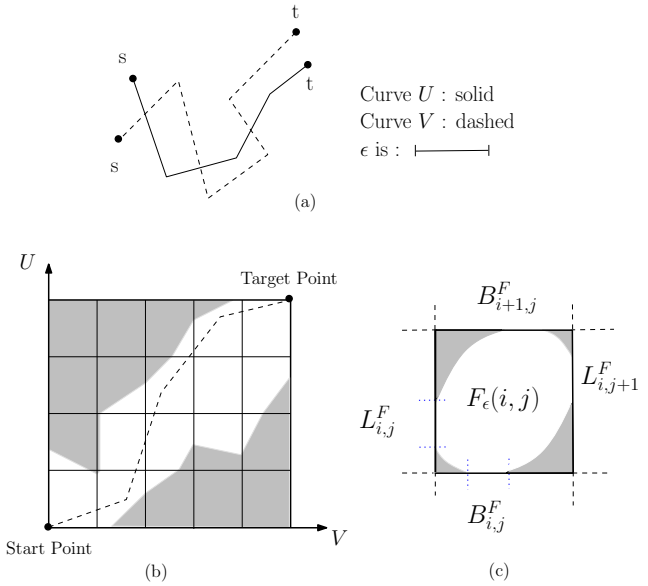

Figure 1: Illustration of the free space diagram. It is drawn using a java applet developed by S. Pelletier.

In our algorithm, as a pre-processing step, all cells in the free space diagram are computed for a given constant $\epsilon \geq 0$. Let O_u and O_v be the point objects moving on U and V, respectively. In order to compute the minimum slope which is permissible in a cell(i,j), O_u must move with Min_i speed and O_v with Max'_j speed. To compute the maximum slope, permissible in a cell(i,j), O_u must move with Max_i and O_v with Min'_j. An *s-monotone path* is a path monotone in both coordinates from the start point to any point in the free space diagram which consists of segments, each of which have permissible slope at the cell passed. Since O_u and O_v move at a constant speed on every segment of their respective polygonal paths, the s-monotone path consists of only straight lines.

It may be tempting to perform a linear transform to each cell so that the s-monotonicity is transformed to x-y monotonicity. But when this is applied to all cells simultaneously in the free space diagram, the transformed cells are not a tiling of $[0,1]^2$ since the speed limits (and hence the slopes) are not the same for all cells.

From now, when we say a point p is reachable, that implies that there is an s-monotone path from the start point to that point in the diagram. We call an interval reachable if all of its points are reachable.

Let the *entry side* of a cell(i,j) be $L^F_{i,j} \cup B^F_{i,j}$ (Figure 1c). Similarly, let the *exit side* of a cell(i,j) be $L^F_{i,j+1} \cup B^F_{i+1,j}$. The *minLine* and *maxLine* of a cell(i,j) are straight lines from the entry side to the exit side of that cell with minimum and maximum permissible slope of a cell(i,j), respectively. Since neither O_u nor O_v are allowed to backtrack, the permissible minimum and maximum slopes are non-negative.

Let $L^S_{i,j}$ denote a line segment on $L^F_{i,j}$ that includes all points on $L^F_{i,j}$ which are reachable from the start point of the diagram. Likewise, let $B^S_{i,j}$ denote a line segment on $B^F_{i,j}$ that includes all reachable points on $B^F_{i,j}$.

We now state our naive algorithm to the decision problem. Cells are processed one by one, row-wise, from the first cell to the last cell in the diagram, to compute L^S and B^S on boundaries of cells. To construct $L^S_{i,j+1}$ and $B^S_{i+1,j}$ from $L^S_{i,j}$ and $B^S_{i,j}$, the left end-points of $L^S_{i,j}$ and $B^S_{i,j}$ are projected with maxLine; the right end-points are projected with minLine to the exit side of a cell(i,j). This projection may create an interval on the exit side of a cell which is unreachable from the start point. We call such an interval a *block*. We denote a new block formed on the exit side of a cell(i,j) by $K_{i,j}$. The left (right) end-point of a block on $L^F_{i,j}$, has the largest (smallest) y-coordinate among all points on that block. Analogously, the left (right) point of a block on $B^F_{i,j}$, has the smallest (largest) y-coordinate among all points on that block. Assume that two blocks K_1 and K_2 lie on the entry side of a cell. We say that K_1 lies to the left of K_2, if the left end-point of K_1 has smaller x-coordinate or smaller y-coordinate than the left end-point of K_2.

In the naive algorithm, all blocks that lie on the entry side of cell(i,j) are projected, one by one, to the exit side of that cell. Since points on a block are not reachable, the left end-point of each block is projected with minLine and the right end-point, is projected with maxLine to the exit side of cell(i,j). If these two projection lines cross each other inside cell(i,j), we say the block *vanishes*, otherwise we say it *survives*. We call a surviving block, a *grey* block, if its projection lies completely in the grey area of a cell (see Figure 2a). As a result of projecting blocks, L^S and B^S of the cells may be fragmented into intervals, some of which are reachable, some are unreachable (see Figure 2b).

During the processing of cells, blocks which vanish and those whose projections entirely lie in grey area of the diagram, are discarded. Thus, only the surviving blocks are maintained. Finally, at the last cell, if projection of $L^S_{u,v}$ or $B^S_{u,v}$, includes the target point, and projection of no block covers the target point, then the answer to the decision problem is YES, otherwise, the answer is NO. In the full version of this paper we have established that there exist an s-monotone path from start to target in the free space diagram if and only if the output of the naive algorithm is YES. Also, we have shown the following:

Theorem 1 *The naive algorithm solves the decision*

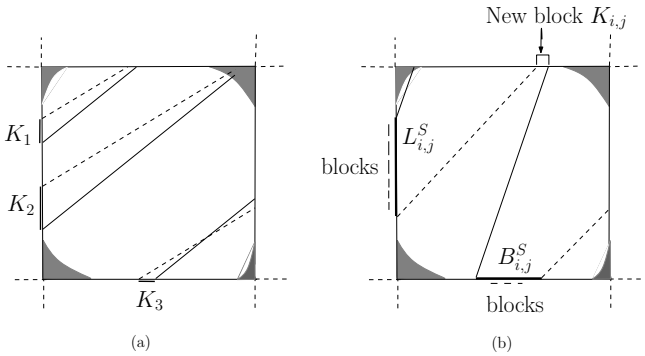

Figure 2: (a) Block K_1 survives after projection, K_2 becomes a grey block and K_3 vanishes (b) A new block is formed on the top of a cell.

problem of Fréchet distance with speed limits correctly in $\Theta(n^3)$ *time.*

3 An $O(n^2 \log n)$ Time Decision Algorithm

In this section, we improve upon the naive algorithm. Rather than projecting all blocks which lie on the entry side of a cell(i,j), we only project the leftmost, the rightmost, and the smallest block. Obviously, this means that some information is not available during the execution of the algorithm. However, we store blocks in a data structure, called *block tree*. We show that we can delay the propagation of the blocks until the blocks are required.

Let s be a line segment on the left (resp., bottom) side of a cell(i,j) and s' be the result of projecting the left end-point of s with minLine, and the right end-point of s with maxLine, to the exit side of that cell. Let the *left (resp., bottom) shrink factor* of cell(i,j) be the ratio $\frac{|s'|}{|s|}$.

First, we formally define the block tree and its operations. Then, we discuss how the block tree is used to propagate reachability information through one cell. Finally, we describe how searching is done at the last cell.

3.1 Block Tree

The block tree is a standard balanced binary search tree built on the relative location of blocks located on the left or bottom entry side of a cell. The notation $T^L_{i,j}$ and $T^B_{i,j}$ are used to denote the left and the bottom block tree of cell(i,j), respectively. For convenience, $T_{i,j}$ refers to $T^L_{i,j}$ and $T^B_{i,j}$. A node in $T_{i,j}$ stores a representative block which appears on the exit side of a cell that is encountered prior to processing the cell(i,j). Because not all blocks are projected, the left and right end-points of the blocks stored at that node are not known. However, our algorithm ensures that all the stored blocks have neither vanished nor have been discarded, and will have a non-empty projection on the entry side of cell(i,j).

During processing, edges of the block tree are labelled with the shrink factors, called β, of the processed cells. These factors are used to compute the exact length of a block stored in a node. A node z in the block tree, corresponding to a block K, stores the following information

1: the left end-point coordinates of K
2: the last cell that the left end-point of K was computed
3: the length of K
4: γ, is a real number which is used in computation of the length of the blocks
5: a pointer to the smallest block in the subtree of z.

Next, we state basic operations on the block tree T.

ExactLocation$(T, z$,cell(i,j)), finds the exact location of the block stored at the node z of T at the entry side of cell(i,j). The corresponding block stored at that node, say it is K, is projected to the cell(i,j), starting from the cell stored at Field 2 of z. Meanwhile, Fields 1 to 4 of z are updated. **Search**$(T, [x_1, x_2])$, this procedure returns all blocks stored in T, in the interval $[x_1, x_2]$. The search starts at the root of T. Branching decision at each node z of T is made using the corresponding block stored at that node. First, the exact location and length of that block at the current cell is computed by calling ExactLocation function. Based on the actual location of this block relative to $[x_1, x_2]$, the search continues down the tree and all blocks, lying in the interval $[x_1, x_2]$ are reported. **Insert**$(T, K_{i,j})$, inserts a new node z into T to store the new block $K_{i,j}$. After insertion of z, rotations, similar to those for balanced binary search tree, are performed to keep the block tree balanced and the shrink factor on edges are updated accordingly.

NextSmallestBlock(T), returns the second smallest block stored in T. **RemoveSmallBlocks**(T, l), eliminates all blocks stored in T of length smaller than a real number l. To find those blocks, l is compared to the length of the smallest block which the root of the tree points to. If the length of the smallest block stored in a node is less than l, that node is removed from the tree. By calling the NextSmallestBlock function, the next smallest block in T, is found and its length is compared to l. Deletion is continued until all blocks, smaller than l, are removed from T. **RemoveGreyBlocks**$(T, [x_1, x_2])$, this procedure removes all blocks stored in T, in the interval $[x_1, x_2]$. Finding those blocks is similar to the Search procedure. **Length**(T, z), computes the exact length of a block stored in a node z of T. The block tree is traversed from the root to the node z. The β values on each edge along the path to z are multiplied with γ stored at z and with the length of its block. This determines the

actual length of the block stored in the node.

Split(T), splits a block tree T into two new block trees T_1 and T_2. Either T_1 or T_2, includes the root of T; say it is T_1. The β-values on all edges of T, along the path from the root of T to the root of T_2 are multiplied. That value is then multiplied with β's on the left and right edge of T_2's root. **Merge**(T_1, T_2), this procedure receives two block trees T_1 and T_2 as input, one is to the left of the other. It merges them and updates the pointers to the smallest intervals to form a new block tree. **Shrink**(T, μ), receives a shrink factor μ and a block tree T and multiplies all β values on the left and the right edge of the root of that tree by μ. Also, γ of the root changes to $\gamma = \gamma \times \mu$. **Copy**$(T_1)$, receives a block tree T_1 and returns a copy. It is used to copy blocks, stored in T_1 on the entry side of a cell to another block tree T_2 on the exit side of the cell. Through this operation, only pointers to the block tree T_1 are copies to T_2. One of the main observations which we have is that all these operations preserve the lengths of the blocks on the entry side of a cell.

3.2 Propagating reachability through a cell

Let $MinLengthToSurvive_{i,j}$ be the minimum length that an interval on the entry side of a cell(i,j) must have to survive after projection. To improve the running time of the naive algorithm, we need to propagate reachability information efficiently for which we use the block tree. $T^L_{i,j}$ and $T^B_{i,j}$ store blocks located on the entry side of a cell(i,j) and the following steps A to G are executed. Steps A to D are performed on the entry side of cell(i,j) while steps E to G are performed on its exit side.

(A) Procedure RemoveSmallBlocks is called with $T_{i,j}$ and $l = MinLengthToSurvive_{i,j}$ as input. Blocks whose lengths are insufficient to survive are removed from $T_{i,j}$.

(B) Procedure RemoveGreyBlocks is called with left and bottom block trees to remove all blocks whose projections lie completely in the grey area of the current cell. As a consequence of removing such blocks, one of the trees may be split into two trees.

(C) Procedure Shrink is called twice: on the left block tree with the cell's left shrink factor and on the bottom block tree with the bottom shrink factor.

(D) The leftmost, rightmost and smallest blocks in $T^L_{i,j}$ and $T^B_{i,j}$ are found and are projected to the exit side of cell(i,j). Analogously to Step B, one of the left or bottom block trees may be split.

(E) The Copy Procedure copies $T^L_{i,j}$ and $T^B_{i,j}$ to construct $T^L_{i+1,j}$ and $T^B_{i,j+1}$.

(F) If a new block appears on the exit side of a cell(i,j), then a new node is inserted into the block tree storing that block.

(G) Execute the Merge Procedure, if required, to merge two block trees on the right or the top side of a cell(i,j).

3.3 Searching at the Last Cell

By transferring s-monotone paths from the entry side to the exit side of each cell in row-wise order, we reach the last cell. There, our algorithm makes the final decision concerning the existence of an s-monotone path in the diagram. When the algorithm reaches cell(u,v), $L^S_{u,v} = [y_0, y_1]$ and $B^S_{u,v} = [x_0, x_1]$ and the block trees $T^L_{u,v}$ and $T^B_{u,v}$ are known. These have information of blocks on the entry side of cell(u,v). We first check the inclusion of the target point in the projection of $B^S_{u,v}$ or $L^S_{u,v}$. Next, we back project the target point to the entry side of the cell(u,v) and let the back projection be (x_s, y_s) (see Figure 3). After that, block tree $T^B_{u,v}$ is searched to find blocks lying in the interval $[x_0, x_s]$. Then, the actual length of these blocks are computed by calling the length function. Then, those lengths are summed up to obtain the total length of blocks in $[x_0, x_s]$. This is compared to the length of the line segment x_0x_s. If it is smaller than the length of x_0x_s, then answer to the decision problem is YES; otherwise, the analogous computation is performed for blocks lying in the interval $[y_s, y_1]$.

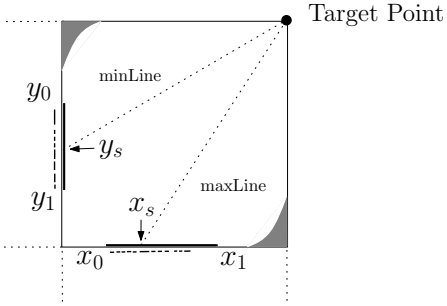

Figure 3: Illustration of the decision procedure at the last cell.

We have shown that propagating reachability through one cell takes $O(\log n)$ amortized time. Furthermore, searching at the last cell, can be shown to take $O(n^2 \log n)$ time, in the worst-case. The correctness and complexity of our algorithm is thus established.

Theorem 2 *Our faster algorithm takes $O(n^2 \log n)$ time and uses $O(n^2)$ space to solve the decision version of the Fréchet distance problem with speed limits.*

By using parametric search, as in [1], the exact Fréchet distance with speed limits can be computed in $O(n^2 \log^2 n)$ time.

References

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