On Practical Nearest Sub-Trajectory Queries under the Fréchet Distance

Joachim Gudmundsson
joachim.gudmundsson@sydney.edu.au
School of Computer Science,
University of Sydney
Australia

Martin P. Seybold
mpseybold@gmail.com
School of Computer Science,
University of Sydney
Australia

John Pfeifer
johnpfeifer@yahoo.com
School of Computer Science,
University of Sydney
Australia

ABSTRACT

We study the problem of sub-trajectory nearest-neighbor queries on polygonal curves under the continuous Fréchet distance. Given a trajectory \( P \) with \( n \) vertices and a query trajectory \( Q \), we seek to report a vertex-aligned sub-trajectory \( P' \) of \( P \) that is closest to \( Q \), i.e., \( P' \) must start and end on contiguous vertices of \( P \).

Since in real data \( P \) typically contains a very large number of vertices, we focus on answering queries exactly, without restrictions on \( P \) or \( Q \), using only pre-computed structures of \( O(n) \) size.

We use three baseline algorithms from straightforward extensions of known work, however they have impractical performance on realistic inputs. Therefore, we propose a new Hierarchical Simplification Tree data structure and an adaptive clustering based query algorithm that efficiently explores relevant parts of \( P \). Experiments on real and synthetic data show that our heuristic effectively prunes the search space and greatly reduces computations compared to baseline approaches.

CCS CONCEPTS

• Information systems → Proximity search; Geographic information systems; Nearest-neighbor search; Clustering.

KEYWORDS

Fréchet Distance, Sub-Trajectory, Nearest-Neighbor, Greedy Decision Algorithm, Hierarchical Simplification Tree, Metric Pruning

1 INTRODUCTION

Increasingly sophisticated and expensive movement capture devices have led to exponential growth in trajectory data over the past decade. Large quantities of trajectory data now exist in diverse fields of study such as sports analysis, human body movement, and vehicle tracking. This has necessitated the need for algorithms that answer trajectory-based queries efficiently. These underlying drivers, along with difficulties in constructing methods that produce exact results, have led to trajectory algorithms \([8, 16]\) that focus on reducing computational complexity by employing approximation algorithms. However, these methods may not be sufficient in settings where meaningful investigation requires exact results.

In this work, we study the problem of exact sub-trajectory nearest-neighbor searches. Given a \( d \) dimensional input trajectory \( P \) of size \( n \), and query trajectory \( Q \) of size \( m \), with \( m \ll n \), the problem is to find a contiguous sub-trajectory \( P' \) within \( P \) that is closest to \( Q \) under the continuous Fréchet distance \([1]\). \( P' \) must start and end on vertices in \( P \), and \( P' \) is inclusion minimal, i.e., it cannot be shortened and still be closest to \( Q \).

The problem of sub-trajectory proximity searches under the continuous Fréchet distance has been studied from a theoretical point of view. In particular, work exists on approximation algorithms for the range counting problem \([8, 16]\) and a data mining problem \([5]\) where one searches for similarly close sub-trajectories within input \( P \). However, the exact search problem is less studied.

We study exact algorithms and turn our attention towards pragmatic methods that prune the search space and reduce computations. Our goal is a practical nearest-neighbor search algorithm which gives results that are correct and exact. In our setting, the solution must: (i) return a nearest-neighbor with no restrictions on \( P \) or \( Q \), (ii) have low pre-processing time, and (iii) use linear storage.

1.1 Contribution and Paper Outline

We present three different baseline algorithms (cf. Section 4) that return exact or approximate vertex-aligned sub-trajectory results, obtained by slight modifications of existing work to our problem setting. These are (B1) adjusting the decision algorithm of Alt and Godau \([1]\), (B2) adapting the approximate Fréchet distance algorithm of Driemel et al. \([11]\), and (B3) using the metric indexing technique from \([15]\). These baseline algorithms have fast query times for certain types of inputs and queries, however all have drawbacks, such as impractically large data structures and query search spaces, on realistic input (cf. Figure 1).

We counter those drawbacks with a new data structure and sub-trajectory query algorithms (cf. Section 5) that are practically more efficient in searching the input space on real and synthetic data sets. The method uses three techniques, with the third expanding on the first two and also offering the best experimental performance.

Our first technique (cf. Section 5.1) is a novel greedy method, with backtracking, that searches the freespace diagram between \( P \) and \( Q \) to answer the sub-trajectory Fréchet decision problem in
Both studies above bound query times, at a cost of quadratic or larger pre-processing time, limitations to $Q$, and approximate results. They also provide counts and do not report sub-trajectory start/end points.

In [5], Buchin et al. study a data mining problem that detects similar 2D sub-trajectories in $P$ under the discrete and continuous Fréchet distance measures. One result is an optimization algorithm that finds the maximum sub-trajectory length $\ell$ where there are at least $\varepsilon$ sub-trajectories and the continuous Fréchet distances between them are at most $\tau$. The algorithm gives a 2-distance approximation and runs in $O(n^3\ell)$ time and $O(n^2\ell)$ space.

Driemel and Har-Peled [10] describe a linear size data structure for $P$, that takes a single segment $Q$ and points $x$ and $y$ as input, and returns a $(1 + \varepsilon)$-approximation Fréchet distance between $Q$ and the sub-trajectory $(p_1, \ldots, p_y)$, in $O(7n \log n \log \log n)$ time. There is also work [12, 13] on approximate nearest-neighbor searches under the discrete and continuous Fréchet distance measures that find the closest trajectory within an input set, however the methods do not search for sub-trajectories and have exponential data structure size.

The recent work [15] contributes a practical approach for exact proximity searches on sets of input trajectories based on clustering with strong quality guarantees and query algorithms that exploit potentially low ‘intrinsic dimensionality’ [17, 18] of the data sets for metric pruning. However, the method does not extend well to our problem since the data structure size is quadratic in the sub-trajectory setting.

3 PRELIMINARIES

We now provide definitions for trajectories, the continuous Fréchet distance, the nearest-neighbor sub-trajectory search problem, and trajectory simplifications.

3.1 Trajectories
An input trajectory $P$ of size $n$ is a polygonal curve through a contiguous sequence of $n$ vertices $(p_1, \ldots, p_n)$ in $\mathbb{R}^d$, where each vertex pair $p_i, p_{i+1}$ is connected by a straight-line segment $\overline{p_ip_{i+1}}$. The length of $P$ is the sum of the Euclidean lengths of all its segments. A query is a trajectory $Q$ of size $m$, $Q = \langle \overline{p_1, \ldots, p_m} \rangle$.

A sub-trajectory of $P$ is denoted $P'$, and is vertex aligned, meaning its first and last vertices are vertices of $P$, i.e. $P' = \langle \overline{p_i, \ldots, p_j} \rangle$ with $1 \leq i \leq j \leq n$.

3.2 Continuous Fréchet (CF) Distance
The continuous Fréchet distance between two trajectories $P$ and $Q$ can be envisaged as the minimum ‘leash length’ required between a person walking monotonously along $P$, and their dog walking monotonously along $Q$. We associate $P$ with its natural parametrization $P : [0, 1] \rightarrow \mathbb{R}^d$, which maps positions relative to $P$’s length to spatial points — e.g. $P(0.5)$ is the mid-point of $P$. A continuous, monotonous map $f : [0, 1] \rightarrow [0, 1]$ is called a reparametrization, if $f(0) = 0$ and $f(1) = 1$, with $F$ representing the set of all reparametrizations. The continuous Fréchet distance is defined as

$$CF(P, Q) = \inf_{f \in F} \max_{g \in \mathcal{G}} \| P(f(\beta)) - Q(g(\beta)) \|,$$  

(1)
where $\| \cdot \|$ is the Euclidean norm in $\mathbb{R}^d$. We refer to the continuous Fréchet distance as $\text{CF}$ or distance throughout this work, when it is contextually clear. $\text{CF}$ can be computed in $O(mn \log mn)$ time using the algorithm of [1], which performs multiple calls to an $O(mn)$ time decision procedure, denoted $\text{DP}(P, Q, \tau)$, that tests if $\text{CF}$ is at most $\tau$.

The continuous Fréchet distance is a metric and hence can be used in metric indexing schemes [3, 15] (technically it is a pseudo-metric but for the purpose of this work it can be considered a metric).

### 3.3 The Nearest Sub-Trajectory Problem

Given an input trajectory $P$ and query trajectory $Q$, with $n \ll m$, the problem is to find a sub-trajectory $P'$ that is closest to $Q$ under the continuous Fréchet distance. Both the sub-trajectory $P'$ and the CF distance between $P'$ and $Q$ must be reported.

In our setting, there can be more than one sub-trajectory of $P$ that is closest to $Q$. In this case, we report the inclusion minimal [8] sub-trajectory, i.e., $P'$ cannot be shortened and still be closest to $Q$. If there is more than one minimal inclusion result, then report one of them.

We also define the concept of inclusion maximal $P'$, which is used in our third query algorithm (cf. Section 5.4). In this case $P'$ cannot be lengthened in size and still be closest to $Q$.

### 3.4 Greedy Ball Simplification

We use the trajectory simplification algorithm by Driemel et al. [11], which takes a simplified trajectory. It is denoted $s(P, \epsilon)$, and works as follows. Record the initial vertex $p_1$, and set it to the current vertex. Scan the next vertices, in order, until the first $p_1$ that is further than $\epsilon$ away from the current vertex. Record $p_1$ and set it to the current vertex. Repeat until reaching $p_m$ and record $p_n$. The recorded vertices are the simplification result of $P$. The algorithm runs in $O(n)$ time, and the simplification result has $\text{CF}(P, s(P, \epsilon)) \leq \epsilon$.

The greedy ball algorithm has useful properties: (i) it snaps all contiguous segments to $p_1$, if they are within $\epsilon$ distance from $p_1$, (ii) the simplified curve always contains the first and last vertices of $P$, and (iii) all simplified segments (except the last one) have length greater than $\epsilon$.

### 4 BASELINE QUERY ALGORITHMS

Existing literature presents algorithms for computing the exact [1] or approximate [10, 11] continuous Fréchet distance or computing the exact [15] or approximate [9, 12] nearest-neighbor on an input trajectory set under the continuous Fréchet distance.

With relatively small and straightforward modifications, some of these algorithms can be modified to solve the problem we study, the vertex aligned sub-trajectory nearest-neighbor search. We discuss three baseline algorithms and then provide realistic examples where they are ineffective. The three baseline algorithms broadly cover known methods, and are the starting point for the design of our novel data structure and algorithm that overcomes baseline performance issues (see Figure 1).

### 4.1 Baseline 1 - Freespace Decider

A relatively straightforward modification to Alt and Godau’s decision procedure [1] yields the first baseline sub-trajectory nearest-neighbor algorithm. The decision procedure algorithm decides if $P$ and $Q$ are at most $\tau$ distance apart: $\text{DP}(P, Q, \tau)$ returns $true$ if $\text{CF}(P, Q) \leq \tau$, otherwise it returns $false$. We first provide a short description of the classic decision procedure algorithm, then explain the modification.

The $\text{DP}$ algorithm computes a freespace diagram [1], which is a grid that shows all pairs of points on $P$ and $Q$ that are at most $\tau$ distance apart:

$$\text{FS}(P, Q) = \{ (s, t) \in [1, m] \times [1, n] \mid \| q_s - p_t \| \leq \tau \},$$

where $s$ and $t$ are points (on vertices or segment interiors). $\text{FS}$ is discretized with $m$ vertical grid lines corresponding to $Q$ vertices, and $n$ horizontal grid lines corresponding to $P$ vertices, with $(q_1, p_1)$ at the bottom-left grid corner, and $(q_m, p_n)$ at the top-right grid corner. There are $(m - 1) \times (n - 1)$ grid cells, each representing two segments, one from $P$ and one from $Q$, and [1] shows that the freespace for a given cell is computed by determining the intersection of an ellipse and the cell edges (runs in constant time since it is convex).

A reachable point $(q_s, p_t)$ in the $\text{FS}$ is defined as a monotone path through freespace from $(q_1, p_1)$ to $(q_s, p_t)$, where points on $P$ and $Q$ continuously increase along the path (e.g., they cannot ‘walk backwards’). Alt and Godau show that if a monotone path can be constructed through reachable space from $(p_1, q_1)$ to $(q_m, p_n)$, then $\text{DP}(P, Q, \tau) = true$, otherwise it is $false$.

An algorithm for computing the reachable space is as follows. $\text{FS}$ grid cells are scanned row-by-row starting at the bottom row, and within a row cells are searched from left-to-right. Reachable space is propagated from $(q_1, p_1)$ along a monotone path, as each cell’s freespace is computed. If for a given grid row, there is no reachable space along the top boundary, then stop and return $false$, else if $(q_m, p_n)$ is reached return $true$. The algorithm has $O(mn)$ runtime and can be implemented in $O(mn \log mn)$ space.

The classic decision procedure algorithm above can be modified to answer the following question: decide if any sub-trajectory $P'$ within $P$ is less than or equal to continuous Fréchet distance $\tau$ from query $Q$: $\text{DP}_{BD}(P, Q, \tau)$ returns $true$, if for any $P'$, $\text{CF}(P', Q) \leq \tau$, otherwise it returns $false$.

Intuitively, one just needs to modify the algorithm above to search for a monotone path from any freespace starting on the left side of the freespace diagram $(q_1)$ to any reachable space ending on the right side of the freespace diagram $(q_m)$. If such a path exists, then it follows there is a $P'$ (that starts on $p_i$ and ends on $p_j$) such that $\text{CF}(P', Q) \leq \tau$.

We modify the classic decision procedure algorithm as follows. Scan the matrix cells column-by-column starting at the left-most column, and within a column cells are searched from bottom-to-top. Propagate reachable space the same as above. If for a given column, there is no reachable space on the right boundary of the column, then stop and return $false$, else if reachable space is propagated to any point on the right side of the freespace diagram $(q_m)$, then return $true$. Note that the search is done column-by-column. The
decision algorithm has $O(mn)$ runtime and can be implemented in $O(n)$ space. In our setting $P'$ is vertex aligned, so vertices $p_i$ and $p_j$ must be in the reachable space, which must be checked in every call to $DP_{FD}$. With the $DP_{FD}$ algorithm above it is trivial to check if one or more end points in $P'$ is in reachable space. To check if one or more start points in $P'$ is in reachable space, search the reachable space in reverse.

Similar to the continuous Fréchet distance computation [1], the sub-trajectory nearest-neighbour algorithm, denoted $FD(P, Q)$, calls $DP_{FD}(P, Q, \tau)$ multiple times on a set of critical values, and can run in $O(n \log mn)$ time. The distance and minimal inclusion sub-trajectory are returned for the case where $DP_{FD}(P, Q, \tau) = \text{true}$ and $\tau$ is minimal.

The Baseline 1 algorithm has two primary issues in our setting, both of which are a consequence of the large size of $P$. The first issue is that the parametric search uses $\Theta(mn)$ space which is unfortunate since in our setting $n$ can be very large. A pragmatic way to avoid this is using a binary search on numeric digits of the distance value until a user-defined precision is reached. The second issue is that the algorithm visits all reachable cells, which can be numerous if $P$ has high sampling density (see Figure 1(a)).

### 4.2 Baseline 2 - Decider on Simplified Curves

Inspired by the algorithm of Driemel et al. [11], we describe a simple method that reports a $(1 + \epsilon)$-approximate nearest sub-trajectory. The main idea is to start with a large simplification error $\gamma$ on $P$ and reduce the error until an approximate result is found. Instead of the binary search on the distance values from a well-separated pair decomposition of the points of $P$ and $Q$ (see Section 3.3.3 in [11]), we use a simple exponential search over the range of possible values for the simplification threshold.

Start with iteration $i = 1$ and set $\gamma = \max\{\text{reach}(P), \text{reach}(Q)\}$, where $\text{reach}(p_1, \ldots, p_n) = \max_j \|p_j - p_1\|$ denotes the maximum Euclidean distance between the start and the other trajectory points [9]. Simplify $P$ to obtain $P' = s(P, \gamma)$. Compute the nearest-neighbor distance $\alpha = FD(P', Q)$, using the Baseline 1 algorithm. If $(\alpha + \gamma)/(\alpha - \gamma) \leq (1 + \epsilon)$ then stop and return the (incluion minimal) sub-trajectory from $FD$. Otherwise set $\gamma$ to $\gamma/2^i$, increment $i$, and repeat on this resolution.

Note that the search for $\alpha$ is particularly demanding for values close to $\gamma(1 + 2/\epsilon) = \tau$. Hence, we use the following criteria to avoid unnecessary precise estimation of $\alpha$ in the $FD$ algorithm. A call $DP_{FD}(P, Q, \tau)$ generates an upper or lower bound on $\alpha$, as discussed in Section 4.1. If $LB(\alpha) \geq \tau$, then stop and return the distance $LB(\alpha)$. If $UB(\alpha) < \tau$ or $[LB(\alpha), UB(\alpha)] \subseteq [\tau/2, 2\tau]$, then stop and proceed to the next finer resolution. The value of $\tau$ in the latter criteria is a heuristic choice; the optimal value depends on the time trade-off between a $DP_{FD}$ call on resolution $\gamma$ and one on resolution $\gamma/2^i$.

The approximation algorithm $SC_{\epsilon}(P, Q)$ runs faster than the Baseline 1 algorithm if there are many sub-trajectories of $P$ that are close. On queries with very small nearest neighbor distances however, the approximation algorithm incurs substantial overhead compared to Baseline 1 (see Figure 1(b) and Section 6.2).

### 4.3 Baseline 3 - CCT Metric Index

Since the Fréchet distance is a (pseudo) metric, one can apply metric indexes for searching the nearest neighbor in the set of sub-trajectories. Metric indexes typically cluster inputs via a metric ball or bisector plane, and examples include the M-Tree [7] which reduces disk I/O accesses, or the Cover Tree [3] whose nearest-neighbor search is bounded in terms of the expansion constant [18].

Fortunately, one can answer sub-trajectory nearest-neighbor queries efficiently with the Cluster Center Tree (CCT) [15], a structure that is specifically designed to cluster and search on trajectories under the CF distance. In our setting, take all $\binom{n}{2}$ pairwise sub-trajectories $P' \in P$ and insert them into the CCT. Then, simply execute the CCT $kNN$ query algorithm for query $Q$ and $k = 1$. The query result will contain the closest vertex aligned sub-trajectory of $P$ to $Q$.

The CCT can provide a more favorable clustering on the set of sub-trajectories compared to our proposed HST, however it has two main drawbacks in our study setting. This first is that the CCT input set contains $\binom{n}{2}$ trajectories, which results in a CCT data structure size of $\Theta(n^2)$. This limitation means that in practical settings one must have input trajectories of small sizes (e.g., $|P| \leq 5000$). Moreover, the CCT construction algorithm may require many CF calls which impacts the pre-processing time. The second drawback is when $P$ has high intrinsic dimensionality, i.e., there are many sub-trajectories in $P$ that are close to $Q$ (see Figure 1(c)). In this case the CCT search algorithm’s pruning is less effective which results in more CF distance computations.

### 5 PROPOSED QUERY ALGORITHMS

This section describes three algorithms for computing the nearest sub-trajectory. The first is a greedy algorithm $DP_{FD}(P, Q, \tau)$ for deciding if $P$ contains a sub-trajectory with a CF distance of at
most \( r \) (cf. Baseline 1). The second algorithm uses the new HST data structure and a breadth-first-search, which extends methods from metric indexing (e.g. the CoverTree [3]) to clusters of sub-trajectories. The third algorithm combines both, which results in a new method that addresses issues highlighted in the Baseline algorithms.

### 5.1 Algorithm 1 - Greedy Decider

There are methods to obtain answers to the decision problem that are practically faster than the aforementioned Dynamic Program of Alt and Godau [1]. Some are based on linear time heuristics that only fall back to the Dynamic Program if the heuristic is inconclusive [2, 6, 13]. Beside those, the work of Bringmann et al. [4] uses a divide-and-conquer approach to construct reachable sections of the freespace, with pruning rules that stop recursions early if a sub-matrix boundary is entirely reachable from the lower left corner or the lower left corner is separated from the upper right corner. However, it is quite unclear if one can modify this approach for the sub-trajectory decision problem.

Our proposed greedy method uses backtracking to solve the decision problem exactly. We first describe the method for deciding if there is a monotone path from the lower left corner \((1, 1)\) to the upper right corner \((m, n)\), and then discuss the modification required for deciding sub-trajectories.

For intuition, imagine there is a metal ball that starts in \((1, 1)\) and that there is a strong magnet above and weaker magnet to the right of the freespace diagram. The magnets attract the metal ball which moves in the freespace and primarily follows the boundary between free and non-freespace. The metal ball’s reachable monotone path is tracked as it attempts to reach \((m, n)\).

We define the canonical path to a (monotonously) reachable point backwards as the path that always chooses the highest reachable predecessor point to reach the last chosen point. The canonical path to \((m, n)\) consists only of sections that are: (i) vertical, (ii) follow a boundary that is non-free above the path, or (iii) horizontal. Note that every end of a horizontal passage coincides with a point on the freespace boundary.

Our decider searches for the canonical path to \((m, n)\) with a sweep over the freespace, i.e. we successively replace some suffix of a canonical path to obtain the canonical path to the next point.

The clockwise (CW) traversal of a boundary curve (between free and non-free space) in any cell partitions its boundary in sections where the traversal is monotonous and non-monotonous (see Figure 2). We use the term stopper for points on the boundary where the traversal changes from monotonous to non-monotonous and trigger for points that switch from non-monotonous to monotonous.

Next, we describe the algorithm in more detail.

#### 5.1.1 Searching from \((1, 1)\) to \((m, n)\)

The algorithm tracks the moving ball in two alternating states. State 1 is monotone movement according to precedence of the magnets, and State 2 is non-monotone movement along the boundary curve (with magnets ‘turned off’).

The algorithm starts in State 1, from point \((1, 1)\), and moves the ball greedily upwards and then rightwards, along a monotonous boundary, until either the target \((m, n)\) or a stopper is reached. Record the State 1 path by appending it to a doubly connected list.

If the monotonous path ends at a stopper \((x, y)\), switch to State 2 and follow the non-monotonous boundary curve in CW direction until: (i) a trigger \((x', y')\) is reached, or (ii) the traversal falls below the floor (height \(y = 1\)). Stop and return false.

If a trigger is reached in State 2, then the goal is to rebuild part of the monotone path by lowering a portion of it. First, find in the currently recorded path the last point whose height is \(y'\), using a naive scan from the end of the list. We call this point \((x'', y'')\) the starter.

Note that a starter is always left of its trigger (i.e. \(x'' < x'\)), and a trigger is always lower than its stopper, i.e. \(y' < y\). Next, scan along a horizontal ray through the freespace, from the starter towards the trigger, which is a temporary target for our search of a canonical path. If the trigger is reached (the ray is not blocked by a boundary), replace the suffix of the recorded path (after the starter point) with the horizontal ray to the trigger, and switch back to State 1. However, if the horizontal ray hits a free space boundary at obstruction point \((u, y')\), which can only be in a non-monotonous section of the boundary (cf. Figure 2 right), then push the pair of starter/trigger onto a stack and repeat State 2 from the obstruction \((u, y')\). A ceiling is set at height \(y'\), which restricts the State 1 traversal to stay below the ceiling.

If at a later point in the State 2 traversal, the vertical line through the ceiling’s trigger point is reached, then pop the starter/trigger off the stack and update the ceiling to the new starter/trigger on the stack top (e.g. point 6 in Figure 3). Note that the trigger points on the stack have monotonously decreasing \(x\)-coordinates. If the stack is empty, the ceiling is just \(n\). For cascading trigger points (each strictly left and not higher than the previous trigger point), the stack keeps track of the cascading ceilings. The ceiling mechanism ensures that the search does not revisit the same stopper point.

Also, note that State 1 obeys the ceiling limit.

See Figure 3 for an example that retrieves the canonical path after only two State 1 traversals.

#### 5.1.2 Analysis

Though the greedy decider may only need to visit a small portion of the reachable cells in practice, its worst-case time exceeds the \(O(mn)\) bound of the well known dynamic program [1].

Our Greedy Decider uses \(O(m+n)\) space since only the trace of the current monotone path is stored in the doubly connected list and trigger points on the stack have monotonously decreasing \(x\)-coordinates. The total amount of work in State 1 is at most \(O(mn(m+n))\), since each of the \(O(mn)\) trigger points is visited at most once and each greedy walk (starting there) takes at most \(O(n+m)\) time. For analyzing the work in State 2, we charge the cost of the naïve suffix scan and the deletion of obsolete elements in the doubly connected list to the creation of the list elements,
i.e. each element only appears once during the suffix scan and the deletion. The non-monotonous traversals in this state are bounded by \( O(mn) \), since every boundary is traversed at most once. Clearly every horizontal scan, from a start to a trigger, takes \( O(m) \) time and there are \( O(mn) \) trigger points. Hence State 2 takes \( O(mn^2) \) time in total and the entire greedy decider takes \( O(mn(m + n)) \) time in the worst case.

5.1.3 Extension for Sub-Trajectories. We now discuss changes required to decide if \( Q \) is within a distance \( \tau \) of any sub-trajectory \( P' \) within \( P \). Similar to the Baseline 1 algorithm, the goal is to search for a monotone path from anywhere on the left side to anywhere on the right side of the freespace diagram. The target is set to any point on the right side (\( x = m \)). Compute the (maximal) freespace intervals on the left side (\( x = 1 \)). Then, for each freespace interval \([y, \hat{y}]\), set the floor to \( y \), start point to \( (1, \hat{y}) \), and begin in State 1. If any of the \([y, \hat{y}]\) intervals gives a canonical path, return \textit{true}, else return \textit{false}. Moreover, when testing the next lower interval at \( x = 1 \) we use the last canonical path from the previous interval to retain the front of pruned space from the freespace. We denote the sub-trajectory greedy decider as \( DP_{GD}(P, Q, \tau) \).

Similar to Baseline Algorithm 1, we check to ensure paths are vertex aligned, and plug \( DP_{GD} \) into an exact sub-trajectory nearest-neighbor algorithm, denoted \( CF_{GD}(P, Q) \), which runs in \( O(mn(m + m) \log mn) \) time. The distance and minimal inclusion sub-trajectory are returned for the case where \( DP_{GD}(P, Q, \tau) = \text{true} \) and \( \tau \) is minimal.

Despite the higher worst-case time complexity, the algorithm may finish very quickly on simple freespace instances, especially given that \( m \ll n \) in our setting. Our experiments show that, on real and synthetic data sets, the number of cell visits rarely exceeds \( n \cdot m \), and that typically only \( O(m) \) cells are visited.

5.2 Hierarchical Simplification Trees (HSTs)

Our proposed HST structure facilitates search space pruning during query execution by enabling the query algorithms to quickly construct sub-trajectory candidates, from coarser to finer trajectory simplifications, based on pre-computed simplifications. Let \( p(i, j) = \text{reach}((p_1, \ldots, p_l)) \) denote the reach of the sub-trajectory between vertex \( p_i \) and \( p_j \) of \( P \). Recall that the greedy ball simplification gives that \( CF(S(c), P') \leq r(l + 1) \).

We call integer \( l \) a \textit{resolution} by associating \( l \) to the ball radius \( r(l) = 2^l \) that is used for the simplification algorithm. E.g. one may think of the set of trajectories \( \{s(P, r(l)) : l \in \mathbb{Z}\} \) as various resolutions of the original trajectory \( P \). For a sub-trajectory, we are interested in the smallest integer whose simplification coincides with the line segment (spanning start and end point).

Nodes in the HST store an interval \([i, j]\), their resolution \( l = \lfloor \log_2 p(i, j - 1) \rfloor \), and a list of children that has either zero or at least two entries. The intervals of the children form a partition of the interval of the parent node and leaves have intervals that contain exactly two consecutive vertices of \( P \) and \( l = -\infty \). For an internal node \( v \), let \( l(v) \) denote its resolution and \( l^+(v) \) the maximum resolution of its children. We call \( v \) active on resolution \( l \) if \( l^+(v) < l \leq l(v) \). See Figure 4 for an example of the structure.

The HST is constructed using the following top-down approach. For the root, store the interval \([1, n]\) and \( l = \lfloor \log_2 p(1, n - 1) \rfloor \). Then recursively refine nodes with \( l = -\infty \) by running the simplification algorithm with \( r(l - 1) \) on the node's sub-trajectory, i.e. \( s((p_1, \ldots, p_l), r(l - 1)) \). For each of the obtained line segments, create one child node that contains the respective interval and compute the resolution of each child. We denote with \( l_b \) the largest and with \( l_h \) the smallest resolution of internal nodes in the HST. Clearly the HST has size \( O(n) \) and depth \( D \leq \min \{|P|, O(\log \Delta(P))\} \), where the spread \( \Delta(P) \) denotes the ratio between the smallest and largest Euclidean distance of the vertices in \( P \).

Building any HST level, based on the previous level, takes \( O(n) \) time for the simplifications and reach computations. Hence the construction time is \( O(nD) \). (See experiments on the obtained node degree and depth in Section 6.2.)
5.3 Algorithm 2 - Finding Trajectory Clusters

The nearest sub-trajectory search in the HST is based on the Cover Tree nearest-neighbor search technique [3], which performs a breadth-first search and prunes branches with the help of clusters and the triangle inequality. However, using the Cover Tree data structure on the set of \( \mathcal{I} \) sub-trajectories is not practical, due to its size and construction time. Instead, for a given HST resolution \( l \), our search uses a method, \( \text{get} \mathcal{C}(l, \mathcal{E}) \), that deduces a set of sub-trajectory clusters on-the-fly based on the clusters \( \mathcal{E} \) from the previous resolution \( l + 1 \). We call such a cluster of sub-trajectories a candidate.

Let \( l(l) \) denote the set of intervals of the HST nodes active in resolution \( l \). A candidate of this resolution is a pair of intervals \( c = ([i, j], [u, v]) \) from \( l(l) \times l(l) \), with \( i < j, u < v \), and \( i \leq u \). We call \([i, j]\) the start interval, \([u, v]\) the end interval, and its associated simplification \( S(c) \) starts at \( p_i \) and ends at \( p_u \), and we have \( CF(S(c), [p_i, \ldots, p_u]) \leq r(l) \). The associated cluster \( C(c) \) of \( c \) consists of the sub-trajectories \( P' = \{p_{s}, \ldots, p_t\} \) with \( s \in [i, j] \) and \( t \in [u, v] \), each of which has distance \( CF(S(c), P') \leq r(l) \) (see Figure 4).

Procedure \( \text{get} \mathcal{C}(l, \mathcal{E}) \) generates, from a candidate set \( \mathcal{E} \) at resolution \( l + 1 \), a new candidate set at resolution \( l \) by replacing those intervals of candidates that are active on \( l + 1 \) but not on \( l \) with one candidate pair for each (newly active) child node.

The basic nearest sub-trajectory search loops over the resolutions of the HST, from \( l \) down to the leaf level. At each resolution \( l \) the following is done. Sub-trajectory candidates are constructed for the resolution based on the remaining candidates from the previous resolution. For each candidate \( c \in \mathcal{E} \), compute \( CF(S(c), Q) \) and set \( \alpha \) to be the smallest of these distances. Then any \( c \in \mathcal{E} \) with \( CF(S(c), Q) > \alpha + 2r(l) \) is discarded from the set of candidates.

Once all resolutions have been searched (\( l = -\infty \)), \( \mathcal{E} \) contains all sub-trajectories of \( P \) that realize the same (nearest-neighbor) distance to \( Q \), e.g. we report the inclusion minimal sub-trajectory if \( |\mathcal{E}| > 1 \).

The HST search is efficient if candidate pruning is effective and candidate sizes \( |S(c)| \) remain small, since this reduces the time to estimate the CF distance. E.g. processing any of the \( \text{get} \mathcal{C}(l, \mathcal{E}) \) candidates of resolution \( l \) takes time \( O(\lambda(l) \log(\lambda(l)m)) \), where \( \lambda(l) = \max\{|S(c)| \mid c \in \text{get} \mathcal{C}(l, \mathcal{E}) \} \) is the size of the longest (encountered) simplification on resolution \( l \). However, Algorithm 2 has the same drawback as the Baseline 3 CCT algorithm. If there are many (exact) nearest sub-trajectory candidates to test, then pruning is less effective and more CF calls are issued (cf. Section 6.2).

5.4 Algorithm 3 - Putting All Together

The improved HST search algorithm overcomes issues mentioned in Algorithm 2 by combining ideas from Algorithm 1 and 2, introducing a switch that changes the search technique if \( |\mathcal{E}| \) gets too large, and incorporating other heuristics that reduce computations in practice.

Algorithm 3 contains two parts. Part 1 (lines 1-7) is a modification of Algorithm 2 that uses faster upper and lower bounds to more quickly search candidates on parent resolutions, and stops if \( |\mathcal{E}| \) gets too large. Part 2 (lines 8-17) searches any remaining parent resolution and the leaf level by calling Algorithm 1, but on (typically) small candidate \( c \) sizes at various simplification resolutions.

**Part 1 - Pruning using Heuristics.** We first discuss Part 1. Line \( 5 \) in Part 1 sets \( \alpha \) to the smallest upper bound \( UB_1(c, Q) \) from [4], which runs in constant time and essentially computes the lengths of \( c \) and \( Q \) to their respective midpoints, and the Euclidean distance between the midpoints of \( c \) and \( Q \), to arrive at an upper bound (see Figure 7 in [4]). Note that a linear time pre-processing of \( P \) gives those sums and stores the Euclidean distance of segments in \( P \). Part 1 (line 6) then discards \( c \) if \( \alpha + 2r(l) \) is less than any of two lower bound computations. The first lower bound \( LB_1(S(c), Q) \) is a constant time bound, also from [4] which uses the same information as the upper bound computation. However, \( LB_1 \) is a very weak bound (e.g., returns \( 0 \)) when the length of \( S(c) \) is long. Therefore, to improve the chance of discarding \( c \), we also call a stronger lower bound, \( LB_2(S(c), Q) \), from [15], which computes the two start/end point Euclidean distances and bounding boxes on \( S(c) \) and \( Q \) in \( O(\log n) \) time using an augmented search tree.

Part 1 (line 4) is the circuit-breaker that checks to see if it is worthwhile or not to continue checking processing parent levels using upper/lower bounds. If the candidate set \( |\mathcal{E}| \) becomes larger than a threshold \( M \) (e.g., a constant), then Part 1 stops early.

If the query on \( P \) is fortunate, meaning the upper/lower bound pruning is effective and the circuit-breaker is avoided, then all parent resolution levels are processed in Part 1 (note that the HST leaf level is always processed in Part 2). Part 1 is also particularly favorable when \( |S(c)| \) is small. Each parent level calls the bounds up to \( M \) times, so the runtime of Part 1 is \( O((l_b - l_h) \cdot M \cdot \log n) \). It uses \( O(n + m) \) space since it must allocate memory for \( max(|c|, |\mathcal{C}|) \).

**Algorithm 3: NearestSubTrajectory(Q)**

**Result:** sub-trajectory distance and \( P' \)
**Data:** \( l \leftarrow l_h; \mathcal{E} = \{(1, n), (1, n)\} \)

\[ \text{while } l > l_h \text{ do } // \text{Part 1} \]
  \[ l \leftarrow l - 1; \]
  \[ \mathcal{E} \leftarrow \text{get} \mathcal{C}(l, \mathcal{E}); \]
  \[ \text{if } l = l_h \text{ or } |\mathcal{E}| > M \text{ then break; } \]
  \[ \alpha \leftarrow \text{smallest } UB_1(S(c), Q) \text{ with } c \in \mathcal{E}; \]
  \[ \text{discard } c \in \mathcal{E} \text{ if } \alpha + 2r(l) < \max(LB_1(S(c), Q), LB_2(S(c), Q)); \]

\[ C \leftarrow \text{aggregateC}(); \]

\[ \text{for } l' \text{ down to } l_h \text{ do } \]
  \[ \alpha, P' \leftarrow \text{smallest } CF_{GD}(S(c), Q) \text{ with } c \in C; \]
  \[ \text{if } l' > l_h \text{ then } \]
  \[ C \leftarrow \text{aggregateC}(C); \]

\[ \text{end} \]

\[ \text{end} \]

\[ \text{Return } \alpha, P' \]
Part 2 - Batching Candidates: We now discuss Part 2 (lines 8-17), which abandons the metric pruning of sub-trajectory clusters inherent in Part 1, and instead opts for a freespace pruning method that calls the proposed Algorithm 1 greedy decider with candidates that have decreasing resolutions.

The first step (line 8) aggregates existing pairwise candidates into a smaller set that is more fortunate for doing searches in the freespace diagram. In aggregateC(\(\mathcal{C}\)), the union of candidate intervals is computed, which combines candidates together that overlap one another (we consider adjacent vertices to be overlapping). The new aggregated candidate set still covers all original intervals, but the aggregation can result in a (much) smaller set of candidates. This ‘batching process’ results in fewer calls to the greedy decider, and avoids unnecessary work since overlapping candidate intervals are eliminated.

The second step (lines 9–16) loops from the HST parent resolution level \(l\) that was stopped at in Part 1 down to the leaf level, and does the following: First, in line 10, for each candidate \(c \in C\), compute \(\text{CF}_{\text{GD}}(S(c), Q)\) and set \(\alpha\) and the sub-trajectory result \(P'\) to the smallest \(\text{CF}_{\text{GD}}(S(c), Q)\), where \(P'\) is the minimal inclusion result. If we are at the leaf level, then the continuous Fréchet distance \(\alpha\) and nearest sub-trajectory \(P'\) are returned. Otherwise, \(\alpha\) plus the resolution error is used as an upper bound to assist in computing a candidate set for the next level down (line 13), which is then aggregated (line 14). We describe candidate set generation in detail below, as well as additional heuristics to speed-up the algorithm.

There are different ways to compute a candidate set for the next lower resolution error (line 13). For example, one could simply discard candidates \(c \in C\) if \(\alpha + r(l') < \text{CF}_{\text{GD}}(S(c), Q) - r(l')\). However, remaining candidates retain their start \([i, j]\) and end \([u, v]\) intervals, even if those parts become further from \(Q\) as the resolution decreases.

Instead, we opt for a candidate set generation method (line 13) that reduces remaining candidate lengths, and prunes away candidates that are too far. This improved pruning method calls \(\text{DP}_{\text{GD}}(S(c), Q, \alpha + r(l'))\) for each aggregated \(c \in C\), which returns maximal inclusion paths, i.e., the new candidates for the next lower resolution level. There are three possible outcomes when calling \(\text{DP}_{\text{GD}}\): (i) \(c\) is pruned since there is no longer a canonical path in the reachable space, (ii) \(S(c)\) is reduced in length since the maximal inclusion path that is returned is smaller in length, or (iii) \(c\) is retained and the maximal inclusion path is not reduced in length. The closest sub-trajectory can be no further than \(\alpha + r(l')\), which is the upper bound, thus ensuring that the candidate with the closest result will always return a maximal inclusion path. As the resolution is reduced in Part 2, the distance \(\alpha + r(l')\) becomes smaller, hence improving the pruning effectiveness when calling \(\text{DP}_{\text{GD}}\). It is important to note that maximal inclusion paths (rather than minimal inclusion) are returned from \(\text{DP}_{\text{GD}}\), to ensure that only non-reachable space that is further than \(\alpha + r(l')\) is pruned.

Three additional heuristics are applied to the \(\text{CF}_{\text{GD}}(S(c), Q)\) call (line 10) in Part 2 when the search is at a parent level:

- Uniformly randomly choose a small number (\(\approx \log |C|\)) of candidates in \(C\) to call \(\text{CF}_{\text{GD}}(S(c), Q)\) and determine \(\alpha\).
- Limit the number of decision procedure calls in \(\text{CF}_{\text{GD}}(S(c), Q)\) to \(2^{(l_l-l_{l-1})}\), and determine \(\alpha\).
- If \(\text{LB}_2(S(c), Q) > \alpha\) or \(\text{DP}_{\text{GD}}(S(c), Q, \alpha) = \text{false}\), then do not check \(\text{CF}_{\text{GD}}(S(c), Q)\).

The first two heuristics do less work nearer the HST root and more work nearer the HST leaf level, and can result in a larger \(\alpha\), thus not violating pruning correctness.

Note that for Part 2, the simplified trajectory \(S(c)\) generated for a given \(c = ([i, j], [u, v])\) starts at vertex \(i\) and ends at vertex \(v\). This differs from the logic in Part 1 which generates \(S(c)\) using end vertex \(u\). This modification ensures that the greedy decider checks all potential \(P\) vertices that may contain the correct result.

5.4.1 Modification for Approximate Nearest-Neighbor. Note that Algorithm 3 is easily adjusted (lines 5 and 10) to allow for results with additive \(\epsilon^a\) or multiplicative \(\epsilon^a\) errors. Pass in \(\epsilon^a\) or \(\epsilon^a\) as part of the query, and if \(\alpha - r(l) > 0\), then do the following check. For additive errors, if \((\alpha + r(l)) - (\alpha - r(l)) \leq \epsilon^a\), then stop and return the \(c\) and \(\alpha\). For multiplicative errors, if \((\alpha + r(l))(\alpha - r(l)) \leq \epsilon^a\), then stop and return the \(c\) and \(\alpha\).

5.4.2 Pruning Effectiveness. It is difficult to choose a satisfactory circuit breaker mechanism to switch from Part 1 to Part 2, since Part 2 can perform much less work than its worst-case runtime on realistic inputs. I.e., the precise tradeoff point between Part 1 and Part 2 is elusive due to heuristics and the grouping of candidates. We set the switch to \(M = 1000\) (line 4) for our query experiments (Section 6), which was determined based on the following experimental evaluation. Nearest query experiments were run on \(|P| = 10^4\) Pigeon and Football real data sets for \(M = 500, 1000, 1500, 2000, 2500, 3000, 3500,\) and \(4000\). \(M = 1000\) gave the best runtime for both data sets. This switch value seems beneficial since it gives Part 1 an opportunity to discard candidates before switching over to Part 2.

Algorithm 3 reduces performance issues associated with the Baseline algorithms. Baseline 1 needs to check all reachable cells, whereas Algorithm 3 uses linear space and heuristics which can greatly reduce freespace cell checks. Baseline 2 performs many loops checking irrelevant parts of \(P\), but Algorithm 3 prunes parts of \(P\) that are too far. Baseline 3 has a quadratic size in the sub-trajectory setting and an issue with multiple exact closest results that require additional distance computations, whereas Algorithm 3 uses the linear size HST, and avoids the multiple result problem by using candidate aggregation.

Algorithm 3 also reduces deficiencies of Algorithms 1 & 2. Algorithm 1’s heuristics can greatly reduce cell-checks, however, it still has to search irrelevant parts of unsimplified \(P\), but the pruning of Algorithm 3 at higher levels of simplification discards unnecessary parts fast and early. Algorithm 3 performs fewer computations than Algorithm 2 since its Part 1 is faster than Algorithm 2.

6 EXPERIMENTATION

We experimentally evaluate the scalability, effectiveness, and efficiency of our proposed algorithms and HST data structure versus three Baseline algorithms. We measure HST construction runtimes and statistics, and for queries we measure runtimes and candidate.
sizes against both real and synthetic 2D data sets. All experiment code and data sets are publicly available.\footnote{See https://github.com/japfeifer/frechet-queries for code and data sets.}

The experiment setup is described first, followed by our results.

6.1 Experiment Setup

Experiments are performed on a laptop computer with an Intel Core i7-10875H CPU and 64GB RAM, using a single threaded Matlab implementation (interpreted programming language) on a Windows 10 64-bit OS. Experiments use a precision binary search algorithm (similar to [4]) for the continuous Fréchet distance (and sub-trajectory version).

Real Data Sets. Two real data sets are used for experimentation. The first data set tracks Homing Pigeons [14], from release sites to a home site, and contains 131 trajectories each having an average of 970.0 vertices. The second data set tracks European Football players on the pitch [19], with trajectories representing the movement of a player when they have possession of the ball, and has 18,034 trajectories, each with 203.4 vertices on average.

Input trajectories $P$ are constructed for $|P| = 500, 10^3, 5 \times 10^3, 10^4, 5 \times 10^4, 10^5$ for each real data set. The assembly of $P$ works as follows. A uniformly randomly chosen trajectory is removed from the data set and appended to initially empty $P$. Then, remove the next trajectory in the data set whose start vertex is the closest Euclidean distance to the current end vertex of $P$, and append it to the end of $P$. This process continues until the desired $|P|$ is achieved.

The real data sets are difficult to search (i.e., many overlapping sub-trajectories that are similar), however, the Football data set is especially challenging as $|P|$ increases since its trajectories are confined to a small area.

Synthetic Data Sets. Two types of synthetic input trajectories $P$ are generated for experimentation. The first type contains an input curve with low intrinsic dimensionality, i.e., it has longer segments and is straighter which results in sub-trajectories that have less overlap. The second type contains an input curve with...
We are interested in further studying our proposed Algorithm 1 (Greedy decider) to determine if additional analysis or changes to the algorithm can improve the worst-case runtime. It would be appealing to match the $O(mn)$ time bound of Alt and Godau’s dynamic program \cite{1}. We are also interested in improving the analysis of proposed Algorithm 2 (Trajectory Clusters) and bounding the runtime based on an intrinsic dimensionality measure such as the expansion constant\cite{18}. The good performance of this algorithm on the Synthetic-low data set implies that the underlying intrinsic dimensionality of $P$ is an important indicator of runtime.

It is also practical to study other heuristics that improve experimental runtimes for our best performing proposed Algorithm 3. For example, a given HST node $v$ at level $l$ can have a resolution error that is (much) less than $r(l)$, so storing and using this information reduces candidate upper bounds which in turn speeds-up discarding candidates.

ACKNOWLEDGMENTS

This work was supported under the Australian Research Council Discovery Projects funding scheme (project number DP180102870).

REFERENCES

\begin{thebibliography}{10}
\end{thebibliography}