A Practical Index Structure Supporting Fréchet Proximity Queries Among Trajectories

Joachim Gudmundsson* Michael Horton† John Pfeifer‡ Martin P. Seybold§

Abstract

We present a scalable approach for range and k nearest neighbor queries under computationally expensive metrics, like the continuous Fréchet distance on trajectory data. Based on clustering for metric indexes, we obtain a dynamic tree structure whose size is linear in the number of trajectories, regardless of the trajectory's individual sizes or the spatial dimension, which allows one to exploit low 'intrinsic dimensionality' of data sets for effective search space pruning.

Since the distance computation is expensive, generic metric indexing methods are rendered impractical. We present strategies that (i) improve on known upper and lower bound computations, (ii) build cluster trees without any or very few distance calls, and (iii) search using bounds for metric pruning, interval orderings for reduction, and randomized pivoting for reporting the final results.

We analyze the efficiency and effectiveness of our methods with extensive experiments on diverse synthetic and real-world data sets. The results show improvement over state-of-the-art methods for exact queries, and even further speed-ups are achieved for queries that may return approximate results. Surprisingly, the majority of exact nearest-neighbor queries on real data sets are answered *without any* distance computations.

Keywords: Fréchet Distance, Dynamic Metric Index, Clustering, Cluster Tree, Cover Tree, Nearest Neighbor, Range Search

1 Introduction

The rapid growth of movement data diversity and acquisition over the past decade poses expanding scalability *and* flexibility demands on information systems. Tracking technologies such as video analysis, RFIDs, and GPS have enabled experts to collect trajectory data on objects as diverse as flying animals [34, 35, 55, 63], shipping vessels [50], basketballs [56], humans [52], vehicles [32, 67, 68], hurricanes [54], athletes [57], terrestrial animals [23, 45], and tablet pen-tip writing [64]. The size of trajectory data sets continues to increase as improved tracking technology records higher frequencies and larger numbers of objects. Real-world data sets [67, 68, 55, 56, 52] consist of tens of thousands trajectories with thousand or more vertices per trajectory and keep growing. Moreover, tracking complex objects whose position consists of several spatial coordinates (e.g. a Bison cow and its calf), challenges researchers to provide *computational* solutions for trajectory data in high dimensions.

A research problem that has recently received considerable attention [8, 10, 25, 26, 28, 37], is the search for efficient data structures and algorithms that enable nearest-neighbor and range queries on large trajectory data sets. Proximity searches are a core engine underlying visualization and classification applications that provide domain-specific researchers with better insight regarding their trajectory

*joachim.gudmundsson@sydney.edu.au
†michael.horton@sportlogiq.com
‡johnapfeifer@yahoo.com
§martin.seybold@sydney.edu.au

data. Example applications are diverse, such as: identifying potential changes in the migration paths of birds [55, 63], locating similar European Football player ball possession trajectories when driving towards the opponent's net [57], determining if shipping vessels stay within range of a shipping path [50], and discovering how many people have a similar commute along a specified route [52].

A challenging task in trajectory data analysis is choosing an appropriate trajectory similarity measure. Common measures include the discrete or continuous Fréchet [7, 15, 17, 18] and Hausdorff [6] distances, which fulfill the triangle inequality, and the non-metric Dynamic Time Warping (DTW) [46] and Longest Common Subsequence (LCSS) [61] similarity measures. We focus on the continuous Fréchet distance for high dimensional trajectory data for a variety of reasons. First, it jointly captures the similarity in the position, shape, and direction between two trajectories. The Hausdorff distance does not capture similarity of directions, which is a requirement for many real-world applications such as human body movement classification. Second, it is less affected by irregularly sampled trajectories and thus suited for simplified trajectories. The latter is particularly useful in practice as real-world data sets are typically simplified in a pre-processing step using standard trajectory simplification algorithms [13, 27, 51, 69]. Third, it is a metric (unlike DTW or LCSS) and hence it can take advantage of metric indexing [40] techniques.

Proximity search problems present difficulties in several regards, which renders asymptotic worst-case analysis often meaningless for concrete instances [53]. In such cases, empirical evidence is especially pertinent to compare solution strategies [40]. For example, real-world trajectory data sets may not contain attributes that lead to worst-case runtimes, but instead behave more 'reasonably' and perform much better in practice. Though we state asymptotic worst case bounds for our algorithms, the evaluation of our proposed solution strategies focuses heavily on a set of robust experiments using a large variety of data sets.

1.1 Related Work

Search problems bound to find k nearest neighbors (kNN) and neighbors within a spherical range (RNN) in vector spaces under a norm have a long and rich history. The well known dD-Tree [11] (a.k.a KD-Tree) successively partitions the input point set $\mathcal{S} \subseteq \mathbb{R}^d$ with alternating axis-orthogonal hyperplanes to obtain a balanced binary tree in the confines of $\mathcal{O}(|\mathcal{S}|)$ space. However, axis-orthogonal range search, using only linear space, requires $\Theta(|\mathcal{S}|^{1-1/d})$ time in the worst-case. The Range-Tree [12] improves this worst-case time with the expense of storage that is exponential in d. This frequent, underlying phenomenon is well known as the 'curse of dimensionality' and Weber et al. [62] show that the *naive scan* outperforms partitioning and clustering techniques for proximity search on average if d exceeds 10. Theoretical and experimental works on general proximity search problems mainly assume that the distance of two elements can be determined in negligible time, e.g. in $\mathcal{O}(d)$ or $\mathcal{O}(1)$. Exact proximity searches on trajectories in $\mathbb{R}^d$ under the continuous Fréchet distance δ_F however are a computationally harder problem than proximity search on mere points of $\mathbb{R}^d$ under Euclidean distances.

Alt and Godau [7] provide an $\mathcal{O}(n^2)$ time algorithm for deciding if the Fréchet distance is at most some given value. Combining this algorithm with Cole's Parametric Search [22] gives an $\mathcal{O}(n^2 \log n)$ time algorithm that determines δ_F. The decision procedure δ_{FD} does not allow strongly sub-quadratic algorithms, unless a common complexity theory conjecture (SETH) fails [15]. Recently, Buchin et al. [18] gave a randomized algorithm that computes δ_F in $\mathcal{O}(n^2(\log\log n)^2)$ time on a word RAM.

Clearly, for exact RNN trajectory queries only δ_{FD} computations suffice, whereas exact kNN queries might well require exact δ_F computations. The 2017 SIGSPATIAL Cup [1] asked for practical data structures to answer RNN queries under δ_F on trajectories in $d = 2$ dimensional space. Top ranked competitors [10, 19, 29] apply filter-&-refine strategies that often use spatial hashing [19, 29] or a quad tree [10] over the trajectory's start point, end point, and bounding box points to determine a potentially smaller list of candidates. Recently Bringman et al. [16] improved further upon their

winning submission with an orthogonal-range search in a $(4d)$D-Tree (i.e. an 8 dimensional KD-Tree) to obtain a candidate result list, which is then refined by heuristic distance computations and an even further tuned decision procedure, to achieve practically fast range queries on three real-world data sets in the plane ($d = 2$).

There is also work on data structures for approximate proximity queries under δ_F. In [26] de Berg et al. present an approximate query structure for kNN and RNN queries. The structure uses $\mathcal{O}(|\mathcal{S}|/\varepsilon^{2\eta})$ space, where $\varepsilon > 0$ is a quality parameter and η the *fixed number* of vertices that every query trajectory Q is restricted to have. The query algorithm returns $S \subseteq \mathcal{S}$ with an additive error of at most $\varepsilon \cdot reach(Q)$ in $\mathcal{O}(1 + |S|)$ time, where $reach(Q)$ denotes the maximum distance from the start vertex of Q to any of its other vertices. Though the structure is dynamic, the vertex number of a query trajectory η must be fixed *prior* to construction and space usage is *exponential* with respect to it. Driemel and Silvestri [28] provide asymptotic analysis on a set of data structures and query algorithms for approximate NN searches under the Discrete Fréchet distance, and even for the Dynamic Time Warping similarity measure. They utilize an asymmetric version of Locality Sensitive Hashing which maps similar trajectories to the same hash table buckets. However the space and queries bounds are exponential in n, i.e. the number of points per trajectory, already for constant factor approximations.

Recently, Xie et al. [65] provided a data structure for performing distributed kNN queries on trajectories using either a 'Discrete Segment Hausdorff Distance' or a 'Discrete Segment Fréchet Distance'. The data structure is constructed by uniformly randomly sampling a set of trajectory segments, which are then used to compute a set of spatial partition boundaries. Within each spatial partition a variation of an R-Tree [39] data structure is constructed by computing the centroid of the bounding box of trajectory segments. Their experiments for exact 10-NN queries under the Discrete Segment Fréchet Distance on a synthetic trajectory data set ($|\mathcal{S}| = 3$M) shows an average run-time of 4.5 seconds, performing $6,000$ distance calls, on a cluster of 16 compute nodes with 152 parallel threads and 512GB total RAM.

There are numerous approaches that seek to extend simple binary serach trees to the proximity search problem for general sets $\mathcal{S}$ under a metric (see Table 9.1 in [40] for a basic overview). Classic *metric tree indexes* partition the input along generalized metric balls or bisector planes, which offer structures using only $\mathcal{O}(|\mathcal{S}|)$ space. Proximity searches attempt to prune sub-trees by means of the query element's distance to a sub-tree representative and the triangle inequality. For example, the static and binary VP-Tree [66] is balanced due to recursively choosing a ball radius, around the picked vantage point, which coincides with the median distance. In contrast, the dynamic and binary BS-Tree [43] recursively partitions elements into the closer of two ball pivots, resulting in a potentially unbalanced tree. The well known M-Tree [21], which is essentially a multi-way BS-Tree, offers strategies to tune I/O disk accesses. None of the above methods provide worst-case guarantees for proximity searches since ball overlap depends on on the underlying input set $\mathcal{S}$. In fact, all pairwise distances can have roughly the same value, which enforces a worst-case query performance of $\Theta(|\mathcal{S}|)$ for all such structures.

More recent approaches build upon clustering ideas to obtain a small set of 'compact' metric balls with little 'overlap' that cover all elements. More formally, for a resolution ε, an ε-net of a finite metric is a set of centers of distance at least ε whose ε-balls cover all elements – e.g. Quadtree cell centers of a certain level. Since packing and covering problems strongly depend on the dimension of Euclidean spaces, authors seek to capture the 'intrinsic dimensionality' of metric spaces for algorithm analysis with measures thereof. Gonzalez' farthest-first clustering [36] provides ε-nets of size no bigger than an optimal $\frac{\varepsilon}{2}$-net, however straight-forward implementations perform $\mathcal{O}(|\mathcal{S}|^2)$ distance calls. Navigating-Nets [49] connect layers of nets, having shrinking resolutions, with additional links for a data-structure, in which the worst-case NN search time can be bounded in terms of the spread and doubling-constant of the finite metric. However, the factor for $|\mathcal{S}|$ in the space bound depends on non-trivial terms over the doubling-constant. The expansion constant γ of [44] is another data set parameter, which is weaker than the doubling constant (c.f. Section 2.2). The Cover-Tree [14] offers a simpler, yet dynamic,

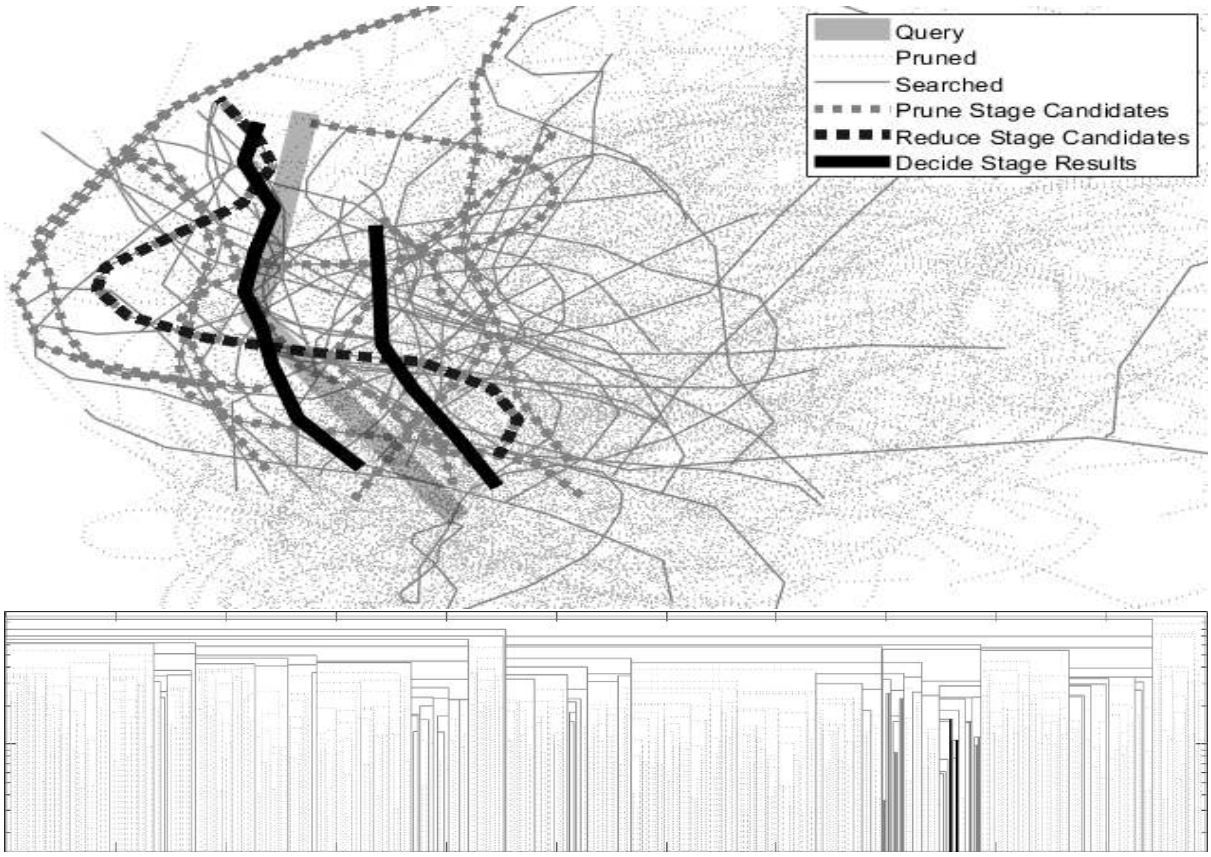

Figure 1: An example of a 2-NN query on 545 bat trajectories [35]. The top plots 2D trajectories: query, pruned, searched, prune stage candidates, reduce stage candidates, and decide stage results. The bottom shows the corresponding CCT dendrogram for nodes that were pruned (dotted line) or searched (solid line) (c.f. Section 5).

approach within the confines of $\mathcal{O}(|\mathcal{S}|)$ space, irrespective of d and 'intrinsic dimensionality' measures of the metric. The authors maintain ε-net properties of tree levels during insert and delete operations, which provides hierarchical cluster trees of arity γ^4 and depth $\mathcal{O}(\gamma^2 \log |\mathcal{S}|)$ whose *form* depend on the expansion-constant γ. Moreover, their NN search tree traversal takes no more than $\mathcal{O}(\gamma^{12} \log |\mathcal{S}|)$ operations. On the other hand, the experiments by Kibriya and Frank [47], on the performance of exact NN search over low dimensional real-world data under Euclidean distances, report a query performance ordering of KD-Trees over Cover-Trees over VP-Trees. The naive scan sporadically outperforms each even on low dimensional real-world data and performances of either method converge on synthetic data with $d \geq 16$, as the curse suggests.

Many real-world trajectory data sets $\mathcal{S}$ consist of ten thousand or more elements and the number of vertices n per trajectory is often in the thousands. Since the performance penalty for a single Fréchet proximity decision δ_{FD} or distance computation δ_F is *huge* (e.g. $n^2 \approx |\mathcal{S}|$ or $n^2 \gg n \log |\mathcal{S}|$), our main objective is to minimize the absolute number of these expensive computations at query time. This is in the same spirit as analysis in the I/O-model [4] of computation, which measures the cost of answering a query as the number of expensive I/O operations performed by the query algorithm. In our setting, the cost is primarily measured in the number of continuous Fréchet distance computation calls performed by the query algorithms.

1.2 Contribution and Paper Outline

We present a scalable and extendable framework for approximate and exact kNN and RNN proximity queries under computationally expensive metric distance functions that is suitable for practical use in

information systems – e.g. proximity queries under the continuous Fréchet distance on high-dimensional trajectory data. In contrast to known approaches, we describe how to effectively extend clustering based, generic metric indexes to dynamic data structures that answer proximity queries correctly but perform only a *very small absolute number* of expensive distance calls. We call this metric index structure Cluster Center Tree (CCT). Using contemporary desktop hardware, our publicly available, single threaded Matlab implementation allows to answer exact proximity queries over a 10M trajectory data set with 1.04 distance calls (latency below 1 second) on average.

	Proposed CCT	Related Work
Data Structure Size	linear	exponential [26, 28, 41]
Construction Time	Variants with $\mathcal{O}(\|\mathcal{S}\|^2)$, but practically fewer, or zero distance calls.	
Query Types	Exact, approximate, and min-error queries for NN, kNN, and RNN under δ_F.	Not for δ_F [28, 41, 65], only approximate [26, 28, 41], only RNN [10, 19, 29], or NN [28, 41] only.
δ_F Calls	Very few in constructions and queries.	Order of magnitude more [21, 14].
Empirical Evaluation	16 real and over 20 synthetic data sets with up to $\|\mathcal{S}\| = 10$M and $d = 32$.	No experiments [28, 41] or few for $d = 2$ only [19, 29, 26, 21, 37, 65].

Table 1: CCTs jointly satisfy many relevant practical aspects whereas related works (c.f. Section 1.1) typically neglect at least one aspect.

Our approach is based on an extendable set of heuristic distance and decision algorithms, which is exchangeable for indexing other computationally expensive metric distance functions. We improve on known heuristic bounds for δ_F and δ_{FD}, which are also practical for high dimensional trajectory data (c.f. Section 3).

Known, generic clustering methods are transferable to CCTs. However, dynamic constructions with $\mathcal{O}(|\mathcal{S}|\gamma^6 \log |\mathcal{S}|)$ distance calls provide coarse cluster radii and static constructions with compactness guarantees use $\mathcal{O}(|\mathcal{S}|^2)$ distance calls. The proposed dynamic and batch construction heuristics achieve CCTs with compact clusters using only very few distance calls – e.g. sub-linear on some instances. Moreover, our approximate radii construction (not excluding exact proximity searches) still achieves compact clusters *without any* distance calls (c.f. Section 4).

We propose heuristic query algorithms that exploit low intrinsic dimensionality in the underlying metric for search space pruning – i.e. excluding clusters of trajectories based on the triangle inequality. To delay unavoidable δ_F and δ_{FD} calls to later stages, our methods leverage cluster compactness and bounds, exclude candidate trajectories based on orderings of the approximation intervals, and finally resolve remaining ambiguity with randomized pivoting for correct query results. Inexpensive heuristic checks further save on some bound computations and our search algorithms naturally extend to queries that may contain approximate results (c.f. Section 5).

Given the aforementioned hardness of exact proximity searches and Fréchet distance computations, we evaluate scalability across various data set characteristics, quality of our CCT constructions, overall query efficiency, and pruning effectiveness with extensive experiments. Observed query performances follow the proposed overlap and compactness metrics for CCT quality. Our experimental results show improvement over recent, state-of-the-art approaches for RNN (even for $d = 2$) and improvement over the generic Cover-Tree, M-Tree and the linear scan (even for $d > 16$). Moreover, the majority of the exact NN queries on our real world-data sets are solved *without any* distance calls and further speed-ups are achieved on approximate queries (c.f. Section 6).

Summarizing aforementioned in Table 1, CCTs jointly satisfy many relevant practical aspects whereas related works (c.f. Section 1.1) typically neglect at least one aspect.

2 Preliminaries

A trajectory P of size m is a polygonal curve through a sequence of m vertices $\langle p_1, \ldots, p_m \rangle$ in $\mathbb{R}^d$, where each contiguous pair of vertices in P is connected by a straight-line segment. Let n denote the maximum size of all trajectories in $\mathcal{S}$. We reserve the term *length* of a trajectory for the sum of the Euclidean lengths of its segments.

Fréchet distance The continuous Fréchet distance $\delta_F(P,Q)$ between two trajectories P and Q can be illustrated as the minimum 'leash length' required between a girl, who walks monotonously along P, and her dog, who walks monotonously along Q. To simplify notation, we associate with a trajectory P its natural parametrization $P : [0,1] \to \mathbb{R}^d$, which maps positions relative to the trajectories length to the spatial points – e.g. $P(0.5)$ is the half-way point. A continuous, monotonous map $f : [0,1] \to [0,1]$ is called a reparameterization, if $f(0) = 0$ and $f(1) = 1$. Let $\mathcal{F}$ be the family of all reparameterizations, then the continuous Fréchet distance is defined as

$$\delta_F(P, Q) = \inf_{f,g \in \mathcal{F}} \max_{\alpha \in [0,1]} \left\| P\Big(f(\alpha)\Big) - Q\Big(g(\alpha)\Big) \right\|,$$

where $\|\cdot\|$ is the Euclidean norm in $\mathbb{R}^d$. We refer to the continuous Fréchet distance as δ_F or *distance* throughout this work, when it is clear from the context. As noted above, most algorithms that compute δ_F base on several calls to an $\mathcal{O}(dn^2)$ time dynamic program which test if δ_F is at most some given value ε. We denote this computation with the predicate $\delta_{FD}(P, Q, \varepsilon)$.

Discrete Fréchet distance The closely related discrete Fréchet distance minimizes over discrete, monotonous mappings $f : \{1, \ldots, m\} \to \{1, \ldots, m\}$ for a trajectory P of size m. It is an upper bound to δ_F, since only alignments of vertex sequences are considered. In fact, the additive error is no more than the length of a longest line-segment in either trajectory (P or Q). Eiter and Mannila [30] gave a quadratic time algorithm, and Agarwal et al. [2] presented a (weakly) sub-quadratic algorithm for computing the discrete Fréchet distance which runs in $\mathcal{O}(mn\frac{\log\log n}{\log n})$ time.

Though DTW differs from discrete Fréchet only in replacing maximum with the summed distances of matched points, the triangle inequality can well be violated on irregular sampled trajectories [1].

2.1 Proximity Search Problems

Our data structure for $\mathcal{S}$ is designed to handle both an additive error $\varepsilon^+ \geq 0$ and a relative error $\varepsilon^* \geq 0$. Though the computer science community prefers the later for algorithm analysis, our interaction with domain experts often leads to additive error specifications. We only state the proximity search problems for the additive error regime, since replacing $+\varepsilon^+$ with $\cdot(1 + \varepsilon^*)$ provides those for the multiplicative.

The k-Nearest-Neighbor Problem:

In: A query trajectory Q, an integer $k \geq 1$ and a non-negative real $\varepsilon^+ \geq 0$.

Out: A set $\mathcal{S}_{k\text{NN}} \subseteq \mathcal{S}$ of k trajectories, such that for all $P \in \mathcal{S}_{k\text{NN}}$ we have

$$\delta_F(P, Q) \leq \tau_k + \varepsilon^+,$$

where τ_k denotes the kth smallest value in the set $\{\delta_F(P, Q) : P \in \mathcal{S}\}$.

[1] The reader may consider DTW among the three 1D trajectories $\langle 0, 2\rangle$, $\langle 0, 1, 2\rangle$ and $\langle 0, 1 - \varepsilon, 1 + \varepsilon, 2\rangle$ as example.

The Range-Search Problem:

In: A query trajectory Q and reals $\tau \geq 0$ and $\varepsilon^+ \geq 0$.

Out: A set $\mathcal{S}_{\text{RNN}} \subseteq \mathcal{S}$ of trajectories, such that both

$$\mathcal{S}_{\text{RNN}} \supseteq \{P \in \mathcal{S} \ : \ \delta_F(P,Q) \leq \tau\} \quad \text{, and}$$
$$\mathcal{S}_{\text{RNN}} \subseteq \{P \in \mathcal{S} \ : \ \delta_F(P,Q) \leq \tau + \varepsilon^+\}$$

hold.

2.2 Intrinsic Dimensionality Measures of Metric Spaces

Let $\mathcal{S}$ be a set and the mapping $\delta : \mathcal{S} \times \mathcal{S} \to \mathbb{R}^+$ a metric on $\mathcal{S}$. For $P \in \mathcal{S}$ we denote with $B(P,\varepsilon) = \{Q \in \mathcal{S} : \delta(P,Q) \leq \varepsilon\}$ the metric ball of radius ε.

Doubling Constant [38] Let $\mu \in \mathbb{N}$ be the smallest number such that for every real $\varepsilon > 0$, every ball in $\mathcal{S}$ of radius ε can be covered by at most μ balls of radius $\varepsilon/2$. More formally, for every $P \in \mathcal{S}$ and $\varepsilon > 0$ there exist $Q_1, \ldots, Q_\mu \in \mathcal{S}$, such that

$$B(P,\varepsilon) \subseteq \bigcup_{i=1}^{\mu} B(Q_i, \varepsilon/2).$$

Expansion Constant [44] Let $\gamma \in \mathbb{N}$ be the smallest number such that

$$\Big|B(P,\varepsilon)\Big| \leq \gamma \Big|B(P,\varepsilon/2)\Big|$$

for every real $\varepsilon > 0$ and $P \in \mathcal{S}$.

We have $\mu \leq 4\gamma$ for finite sets $\mathcal{S}$ (see e.g. Proposition 1.2 in [38]).

2.3 González Clustering for Metric Spaces

Our batch construction algorithms (c.f. Section 4.1) are based on the following farthest-first algorithm for hierarchical, divisive clustering [36]. Given a metric δ on a set $\mathcal{S}$, the algorithm successively adds new cluster centers to a set L.

González-Clustering ($\mathcal{S}, \delta$):

Arrays $\text{dist}[\,] = \infty$ and $\text{parent}[\,] = \emptyset$

1. Pick $C \in \mathcal{S}$
2. Set $L = \{C\}$, $\mathcal{S} = \mathcal{S} \setminus \{C\}$
3. FOREACH $X \in \mathcal{S}$ with $\delta(X,C) < \text{dist}[X]$
   Set $\text{dist}[X] = \delta(X,C)$ and $\text{parent}[X] = C$
4. Pick $C = \underset{X \in \mathcal{S}}{\text{argmax}}\, \text{dist}[X]$
5. Set $L = L \cup \{C\}$ and $\mathcal{S} = \mathcal{S} \setminus \{C\}$
6. If $\mathcal{S} \neq \emptyset$ GOTO 3

Group	Bound	Novelty	Output	Time	d
LB_{F}	$\mathrm{LB}_{\mathrm{SEV}}$	Known	$\mathbb{R}$	$\mathcal{O}(d)$	all
	$\mathrm{LB}_{\mathrm{BB}}$	Improved	$\mathbb{R}$	$\mathcal{O}(d^2 2^{d-1})$	$d \leq 3$
				$\mathcal{O}(d)$	$d > 3$
	$\mathrm{LB}_{\mathrm{ST}}$	New	$\mathbb{R}$	$\mathcal{O}(1)$	all
$\mathrm{LB}_{\mathrm{FD}}$	$\mathrm{LB}_{\mathrm{TR}}$	New	$true/false$	$\mathcal{O}(d(n+m))$	all
UB_{F}	$\mathrm{UB}_{\mathrm{BB}}$	Improved	$\mathbb{R}$	$\mathcal{O}(2^{2d})$	$d \leq 2$
				$\mathcal{O}(d)$	$d > 2$
	$\mathrm{UB}_{\mathrm{ADF}}$	Improved	$\mathbb{R}$	$\mathcal{O}(d(n+m))$	all

Table 2: Overview of bounds and their time complexity for varying dimensions d (c.f. Section 3).

This algorithm requires no more than $\mathcal{O}(|\mathcal{S}|^2)$ distance computations. The following statements on the algorithm's result quality, in terms of minimum cluster number $N(\mathcal{S}, \varepsilon)$ of a ε-cover and minimum cluster size $R(\mathcal{S}, k)$ of a k-center clustering, are well known [36]. To simplify notation, we use for subsets $\mathcal{A} \subseteq \mathcal{S}$ the abbreviation $\delta(P, \mathcal{A}) = \min_{Q \in \mathcal{A}} \delta(P, Q)$ in the following formal definition:

$$R(\mathcal{S}, k) = \min_{\mathcal{A} \in \binom{\mathcal{S}}{k}} \max_{P \in \mathcal{S}} \delta(P, \mathcal{A})$$
$$N(\mathcal{S}, \varepsilon) = \min_{\mathcal{A} \subseteq \mathcal{S}} \left\{ |\mathcal{A}| : \delta(P, \mathcal{A}) \leq \varepsilon \quad \forall P \in \mathcal{S} \right\}$$

Cluster Size and Cover Number Let $C_1, \ldots, C_n$ denote the sequence in which the elements were added to L and let $L(\varepsilon) = \{C \in L : \mathrm{dist}[C] > \varepsilon\}$. We have

$$\begin{aligned} R(\mathcal{S}, k) \leq \quad & \mathrm{dist}[C_k] \quad \leq 2R(\mathcal{S}, k) \qquad \forall k > 1 \\ N(\mathcal{S}, \varepsilon) \leq \quad & |L(\varepsilon)| \quad \leq N(\mathcal{S}, \varepsilon/2) \qquad \forall \varepsilon > 0. \end{aligned}$$

The main observation to prove these statements is the following algorithm invariant: At all times $\varepsilon > 0$, any two elements in $L(\varepsilon)$ have distance of more than ε. Hence, no metric ball of radius $\varepsilon/2$ can cover more than one element of $L(\varepsilon)$, which shows the Cover Number bounds. To show the Cluster Size for some k, one observes that any two elements in $\{C_1, \ldots, C_{k+1}\}$ have distance of at least $\mathrm{dist}[C_{k+1}] =: r$. Hence an optimal clustering with k centers has to contain at least one cluster of radius $r/2$ (see e.g. [24]).

On metrics with bounded doubling constant μ, we additionally have $N(\mathcal{S}, \varepsilon/2) \leq \mu \cdot N(\mathcal{S}, \varepsilon)$ for every $\varepsilon > 0$. This is a key ingredient for the use of 'intrinsic dimensionality' in the analysis of nearest neighbor searches with Navigating-Nets [49], since refining the resolution of an optimal ε-net by a constant does not increases the number of clusters by more than a constant.

3 Fréchet distance bounds

This section describes several fast algorithms for computing upper and lower bounds on the continuous Fréchet distance between two trajectories. These distance approximations are used to speed up the construction of the data structure (Section 4.1) and the query algorithms (Section 5).

Table 2 contains an overview of the bounds together with their time complexities. There are three groups of bounds: (i) a lower bound group LB_{F} (maximum of its bounds), (ii) a lower bound decision procedure $\mathrm{LB}_{\mathrm{FD}}$, and (iii) an upper bound group UB_{F} (minimum of its bounds). The bound groups are applied in the construction and query algorithms.

Given two trajectories $P = \langle p_1, \ldots, p_n \rangle$ and $Q = \langle q_1, \ldots, q_m \rangle$ in $\mathbb{R}^d$, the aim of the algorithms below is to quickly compute upper and lower bounds on $\delta_F(P, Q)$.

3.1 Start and End Vertices (SEV)

Lower bound. A trivial lower bound on the distance between P and Q is the maximum of the Euclidean distances between start vertices p_1 and q_1, and between end vertices p_n and q_m [10, 29, 19]. That is, $\text{LB}_{\text{SEV}}(P, Q) = \max\{\|p_1 - q_1\|, \|p_n - q_m\|\}$, and it can be computed in $\mathcal{O}(d)$ time.

3.2 Axis-aligned Bounding Box (BB)

Let $\text{BB}(P)$ denote the minimum-size d-dimensional axis-aligned box that contains all the vertices of P. It can be computed in $\mathcal{O}(dn)$ time, and, similarly, $\text{BB}(Q)$ can be computed in $\mathcal{O}(dm)$ time.

Lower bound. For $d > 3$ we use a lower bound described by Dütsch and Vahrenhold [29] and Baldus and Bringmann [10]. It computes the maximum of the following as a lower bound: the difference between the maximum x_i-coordinates of $\text{BB}(P)$ and $\text{BB}(Q)$ for each $1 \leq i \leq d$, and the difference between the minimum x_i-coordinates of $\text{BB}(P)$ and $\text{BB}(Q)$ for each $1 \leq i \leq d$. The running time of their algorithm is $\mathcal{O}(d)$.

For $d \leq 3$ we use a different algorithm to that in [29, 10] which can result in a stronger lower bound on $\delta_F(P, Q)$. Let f be an edge (1-face) of $\text{BB}(P)$ and let f' be the corresponding edge of $\text{BB}(Q)$, then $\lambda(f, f')$ is the minimum Euclidean distance, which may or may not be the perpendicular distance (e.g. Figure 2a). Compute the maximum $\lambda(f, f')$ for all corresponding edges of $\text{BB}(P)$ and $\text{BB}(Q)$, which is clearly a lower bound on the Fréchet distance. The number of edges of a d-dimensional bounding box is $d2^{d-1}$, hence the running time is $\mathcal{O}(d \cdot d2^{d-1})$. The lower bound BB algorithm for $d \leq 3$ is denoted $\text{LB}_{\text{BB1}}(P, Q)$, and the algorithm in [29, 10] for $d > 3$ is denoted $\text{LB}_{\text{BB2}}(P, Q)$.

Upper bound. For $d \leq 2$ we use the algorithm by Dütsch and Vahrenhold [29], which computes the maximum of all pairwise distances between the vertices of $\text{BB}(P)$ and $\text{BB}(Q)$. Since the running time of the above algorithm is $\mathcal{O}(2^{2d})$ we use the following modification for $d > 2$. Compute a bounding box that contains all points of P and Q, denoted $\text{BB}(P, Q)$. An upper bound is the Euclidean distance between two vertices of $\text{BB}(P, Q)$, with the first vertex composed of minimum coordinate values for each dimension d, and the second vertex composed of maximum coordinate values for each dimension d. The running time of this algorithm is $\mathcal{O}(d)$, though the upper bound in [29] is slightly stronger. The upper bound BB algorithm in [10] for $d \leq 2$ is denoted $\text{UB}_{\text{BB1}}(P, Q)$, and the algorithm for $d \geq 3$ is denoted $\text{UB}_{\text{BB2}}(P, Q)$.

Rotation. We can further improve LB_{BB1} and UB_{BB1} for trajectories that do not have a directional spine (direction of maximum variance on the point set) that aligns closely with an axis direction. Typical examples of such trajectories can be found in some of the real-world data sets [34, 56, 57] used in Section 6. To obtain a stronger bound for these cases pre-process two other bounding boxes for each input trajectory P by rotating P $22.5°$, and $45°$ counter-clockwise around the origin. At query time, compute the $0°$, $22.5°$, and $45°$ rotation bounding boxes for a query trajectory Q only once. Then, choose the maximum or minimum result from each of the three rotations as the lower or upper bound, respectively. Rotated trajectories can result in a smaller BB and a stronger bound (e.g. Figure 2b). The rotations of $22.5°$, and $45°$ are heuristic values.

3.3 Simplified Trajectory (ST)

Lower bound. Let P' be the straight-line segment between p_1 and p_n and let Q' be the straight-line segment between q_1 and q_m. We set $\text{LB}_{\text{ST}}(P, Q) = |\delta_F(P, P') - \delta_F(Q, Q')|/2$, which we next show is a lower bound for $\delta_F(P, Q)$.

Theorem 1. *$LB_{ST}(P, Q) \leq \delta_F(P, Q)$.*

Proof. From the triangle inequality,

$$\delta_F(Q, Q') \leq \delta_F(Q, P) + \delta_F(P, P') + \delta_F(P', Q') \iff$$

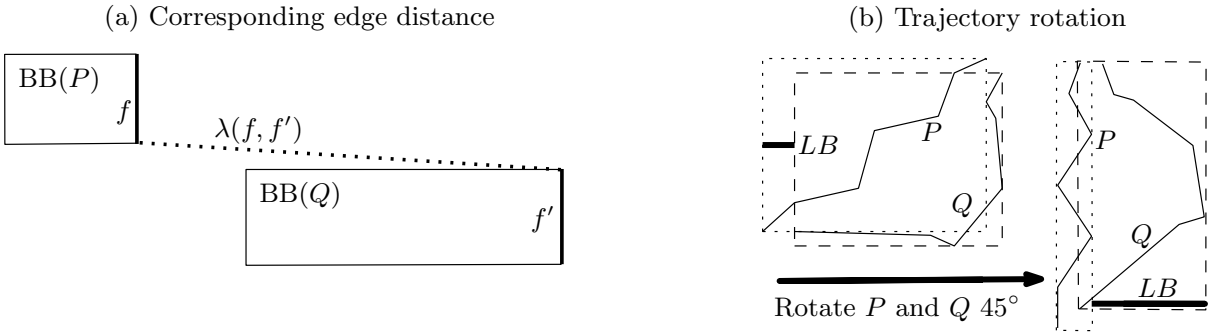

Figure 2: Bounding box lower bound: (a) corresponding edge distance, and (b) trajectory rotation resulting in a stronger lower bound (c.f. Section 3.2).

$$\delta_F(Q,Q') - \delta_F(P,P') \leq \delta_F(P',Q') + \delta_F(P,Q) \leq 2\delta_F(P,Q),$$
$$since \;\; \delta_F(P',Q') \leq \delta_F(P,Q).$$

A similar argument can be used for $\delta_F(P,P')$, hence $|\delta_F(P,P') - \delta_F(Q,Q')|/2 \leq \delta_F(P,Q)$. □

To use this bound pre-compute $\delta_F(P,P')$ for each input trajectory $P \in \mathcal{S}$, in $\mathcal{O}(n \log n)$ time (since P' is a single segment). At query time, once $\delta_F(Q,Q')$ is computed in $\mathcal{O}(m \log m)$ time, then every $\text{LB}_{\text{ST}}(P,Q)$ check for the same query Q is computed in constant time.

3.4 Traversal Race (TR)

Lower bound. Our decision procedure $\text{LB}_{\text{TR}}(P,Q,\alpha)$ for $\alpha \geq 0$, is similar to the negative filter algorithm by Baldus and Bringmann [10]. The algorithm starts at the beginning of P and Q and iteratively traverses P's vertices and Q's edges towards their respective ends. To simplify presentation, we add a first edge $\overline{q_1 q_1}$ and a last edge $\overline{q_m q_m}$ to Q. If the minimum Euclidean distance between P's vertex and Q's edge is less than the given α, then advance to P's next vertex, else advance to Q's next edge. If the end of Q is reached first, then $\alpha < \delta_F(P,Q)$ and answer $true$, otherwise we have not gained any information and answer $false$.

This algorithm gives a stronger bound than the algorithm in [10], especially when the edges of the trajectories are long. Since the algorithm is not symmetric, we run it a second time with P and Q swapped which gives a total runtime of $\mathcal{O}(d(n+m))$.

3.5 Approximate Discrete Fréchet (ADF)

Upper bound. The discrete Fréchet distance is known to be an upper bound on the continuous Fréchet distance [30]. A greedy algorithm in [17], denoted $\text{UB}_{\text{ADF1}}(P,Q)$, approximates the discrete Fréchet distance between two trajectories P and Q in $\mathcal{O}(d(n+m))$ time. The approximation algorithm traverses the vertices of P and Q iteratively from start to end, starting at $i := 1$ and $j := 1$, and at each step picks a pair $(i',j') \in \{(i+1,j),(i,j+1),(i+1,j+1)\}$, minimizing the Euclidean distance between vertices $p_{i'}$ and $q_{j'}$. It holds that $\delta_F(P,Q) \leq \text{UB}_{\text{ADF1}}(P,Q)$ [17].

We include two more variations of the above algorithm. The first, $\text{UB}_{\text{ADF2}}(P,Q)$, traverses the vertices of P and Q in *reverse* from end to start, starting at $i := n$ and $j := m$, and at each step looks backwards to pairs $(i',j') \in \{(i-1,j),(i,j-1),(i-1,j-1)\}$, instead. The second, $\text{UB}_{\text{ADF3}}(P,Q)$, traverses the vertices of P and Q from start to end, starting at $i := 1$ and $j := 1$, and at each step, if $n \geq m$ then increment i and set $j := \lceil m/n \cdot i \rceil$, otherwise increment j and set $i := \lceil n/m \cdot j \rceil$.

We also tried padding trajectories with a small number of new vertices along each edge of P and Q in an attempt to strengthen the bound. However, the experiments showed that this approach very rarely gave any improvements.

Order	If condition is $true$	Return
1.	$\mathrm{UB}_{\mathrm{F}}(P, C_2) \leq \mathrm{LB}_{\mathrm{F}}(P, C_1)$	C_2
2.	$\mathrm{UB}_{\mathrm{F}}(P, C_1) \leq \mathrm{LB}_{\mathrm{F}}(P, C_2)$	C_1
3.	$\mathrm{LB}_{\mathrm{FD}}\big(P, C_1, \mathrm{UB}_{\mathrm{F}}(P, C_2)\big)$	C_2
4.	$\mathrm{LB}_{\mathrm{FD}}\big(P, C_2, \mathrm{UB}_{\mathrm{F}}(P, C_1)\big)$	C_1
5.	$\delta_F(P, C_1) < rad(C_1)/2$	C_1
6.	$\delta_F(P, C_1) < \mathrm{LB}_{\mathrm{F}}(P, C_2)$	C_1
7.	$\delta_F(P, C_1) > \mathrm{UB}_{\mathrm{F}}(P, C_2)$	C_2
8.	$\mathrm{LB}_{\mathrm{FD}}\big(P, C_2, \delta_F(P, C_1)\big)$	C_1
9.	$\delta_{\mathrm{FD}}\big(P, C_2, \delta_F(P, C_1)\big)$	C_2
10.	otherwise	C_1

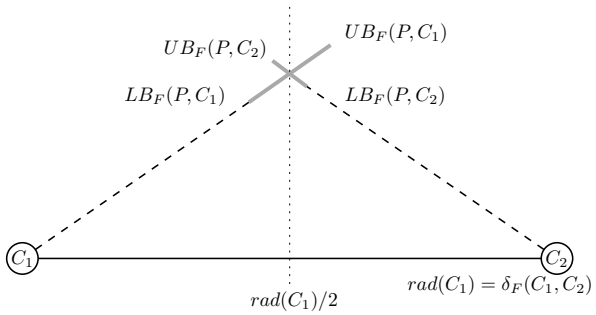

Figure 3: Bisector Localization Predicate for determining if trajectory P is closer to center C_1 or C_2 (c.f. Section 4.1). Subsequent checks are only performed if current results are inconclusive. Test 5 is only performed for the Relaxed CCT since C_2 is a furthest trajectory in the cluster of C_1.

4 Indexing Expensive Metrics with Cluster Center Trees

A Cluster Center Tree (CCT) for a set $\mathcal{S}$ of trajectories is a rooted tree whose nodes represent clusters, that are metric balls of a certain distance radius. Each node v of a CCT stores a distance value $rad(v)$, a reference to some trajectory $C(v)$ (its center), and a list of child nodes. Every trajectory $P \in \mathcal{S}$ appears as the center of a leaf in the tree. An internal node v of a CCT, needs to uphold two properties, which are (*Nesting*) one of its children refers to the same center as v, and (*Bounding*) every descendant u of v has $\delta_F\big(C(u), C(v)\big) \leq rad(v)$. Since the number of leafs is $|\mathcal{S}|$ and each internal node has at least two children, CCTs have a storage consumption within $\mathcal{O}(|\mathcal{S}|)$, regardless of trajectories' size n and dimensionality d.

The following describes three CCT batch construction algorithms (Exact, Relaxed, Approximate) and two dynamic insert/update/delete algorithms (Exact, Approximate), as well as a third insert algorithm (Standard) that similar common dynamic tree indexes use (e.g. the M-Tree [21]).

4.1 Batch CCT Construction

Our batch construction methods are inspired by González' hierarchical, divisive clustering for metric spaces to derive compact clusters (c.f. Section 2.3). Starting with one arbitrary element as the center, the algorithm successively picks an element, as an additional center, that is 'farthest' from any of the previous centers, and then reassigns elements to the additional center if it is closer. A k-center clustering is produced in $k-1$ phases of distance computations and center reassigning. In each phase, the current cluster radii are within a factor of 2 of an optimal k-center clustering that covers all elements (c.f. Section 2.3).

Our construction heuristics foremost aim to avoid or reuse δ_F calls by applying upper and lower bound computations.

4.1.1 Exact CCT Construction

To obtain a binary CCT from the González clustering algorithm in Section 2.3, we consider it an continuous process within the monotonously decreasing radius parameter ε. In addition to the leafs $L(\varepsilon)$, we also track a set of tree nodes $T(\varepsilon)$. Initially, T contains only the root node which is associated to the sole trajectory C_1 in L as its center. Note that the array $\mathrm{parent}[\cdot]$ always points to a leaf for remaining elements in $\mathcal{S}$.

Now, whenever a new center C_i is picked and added to the leaf nodes, we perform a split of its node in T. That is, we replace the leaf's node v that is currently associated to $\mathrm{parent}[C_i]$ in T with a node that points to two children v_1 and v_2, which we associate with the leafs $\mathrm{parent}[C_i]$ and C_i. To reduce the number of distance computations when determining if a given P is closer to C_i or its current center, we use the sequence of bound computations in Figure 3.

After the tree is built, we compute the cluster radii of the CCT in a bottom-up fashion from each leaf. To save δ_F distance calls, we use upper and lower bounds arrays instead of the $\mathrm{dist}[\cdot]$ array and sharpen approximations with δ_F calls only if selecting a furthest element is indecisive. We use the following *Fix-Ancestor-Radius* logic to save on δ_F calls. First check the current radius against UB_{F}, then check against it with $\mathrm{LB}_{\mathrm{FD}}$ and then δ_{FD}. Only if these checks are indecisive, compute δ_F to update the radius of the node's parent.

The worst-case number of distance calls is $\mathcal{O}(|\mathcal{S}|^2)$, since (i) on every iteration all bounds may fail to be conclusive and distances are computed for all trajectories $P \in \mathcal{S}$, (ii) the CCT may degrade to a linear chain on metrics with large spread and asymmetric clusters (e.g. all trajectories are single, 1D points with coordinates of the form $2^i \in \mathbb{R}$), and (iii) Fix-Ancestor-Radius logic may perform up to quadratic δ_F calls. However, our experimental data (Figure 14) shows that this method performs far fewer δ_F calls on real data sets.

4.1.2 Relaxed CCT Construction

This recursive construction algorithm successively performs only one phase of the González algorithm that results in a partition of the trajectories via the metric bisector of the two clusters' centers. This essentially omits the trajectory reassigning in González' clustering.

Pick an arbitrary trajectory $P \in \mathcal{S}$ as center of the root node v, that is $C(v) := P$, and let $\mathcal{S}(v) = \mathcal{S}$ denote the trajectories contained in the cluster of v. The recursive split then determines a trajectory $F(v)$ which is furthest from $C(v)$, which also determines $rad(v)$. To do this, we first compute the highest lower bound α to the distances of $C(v)$ and elements of $\mathcal{S}(v)$. Then we compute δ_F only for those trajectories whose upper bound distance (to $C(v)$) exceed α.

The cluster is then partitioned into (potentially) smaller clusters v_1 and v_2, which are the children of v. For their centers, we set $C(v_1) := C(v)$, $C(v_2) := F(v)$ and assign each trajectory $P \in S(v)$ to the sub-cluster of the closer center. To reduce the number of distance computations when determining if P is closer to $C(v_1)$ or $C(v_2)$, we use the test sequence in Figure 3.

The worst-case number of distance calls is again $\mathcal{O}(|\mathcal{S}|^2)$, since the algorithm may need to compute $\mathcal{O}(|\mathcal{S}|)$ distances at each level of the tree. However, experimental results in Figure 14 shows that this method typically allows one to build CCTs with $\mathcal{O}(|\mathcal{S}|)$ distance calls.

4.1.3 Approximate CCT Construction

Since distance calls are very expensive, we also describe a construction algorithm that performs no calls at all to δ_F and δ_{FD}, that originates from adapting the Relaxed construction. For this, we only use upper bound computations UB_{F} to determine the furthest trajectory $F(v)$ and we assign P to the center, i.e. $C(v_1)$ or $C(v_2)$, that realizes a smaller upper bound value. Compared to the Relaxed method, the approximate method does not perform expensive distance calls but the cluster radii are potentially larger.

4.2 Dynamic CCT Constructions

Given the few properties CCTs need to uphold, there are several heuristic strategies to handle dynamic situations.

4.2.1 Exact Dynamic Inserts

Exact inserts *may* perform distance computations, since cluster radii values are computed exactly.

A new trajectory P is inserted by first locating the leaf v_1 that is an *exact* nearest neighbor of P (c.f. Section 5.1.1). Then we create two new leaf nodes u_1 (contains trajectory of v_1) and u_2 (contains P), and point v_1 to the new nodes. Then fix the radius of v_1 and its ancestors using the already discussed 'Fix-Ancestor-Radius' bottom-up process.

The worst-case number of distance calls is $\mathcal{O}(|\mathcal{S}|)$, since 'Fix-Ancestor-Radius' may need to compute the distance for every tree node. Hence, constructing a CCT entirely with dynamic inserts requires $\mathcal{O}(|\mathcal{S}|^2)$ distance computations. However, our experiments show that the number of distance calls is much smaller for our data sets (c.f. Figure 14).

4.2.2 Approximate Dynamic Inserts

Approximate inserts perform *no* distance computations, and cluster radii are computed based on the largest upper bound value.

A new trajectory P is inserted by first locating the leaf v_1 that is an *implicit approximate* nearest neighbor of P (c.f. Section 5.3). Then we create two new leaf nodes u_1 (contains trajectory of v_1) and u_2 (contains P), and point v_1 to the new nodes. Then fix the radius of v_1 and its ancestors by only checking the current radius against UB_F.

4.2.3 Standard Dynamic Insert

A classic insertion method for metric tree indexes [58, 59, 43, 40] is to start at the root and descend to the child node whose center is closest to new trajectory P, until a leaf v is reached. We adapt this algorithm for our setting by descending to the child node with the closest LB_F to locate leaf v, and then proceed with the same logic as the approximate insert above.

4.3 CCT Quality Analysis

To gain insight of the CCT quality achieved by the various batch and insert algorithms, refer to Figure 4 (see Section 6 for the complete experimental setup).

The average leaf depth is more balanced for the insertion algorithms compared to the batch construction algorithms. However, it is noteworthy that tree depth is inversely proportional to the performance of the construction and query algorithms (see Figure 14 in Section 6.2.1). E.g. unbalanced CCTs do not necessarily incur poor query performance. This may seem counter-intuitive at first, but surveys have mentioned that this can occur [40], and the next two CCT quality measures help to explain why.

The compactness measure tends to be largest for the standard insert and smallest for the exact batch construction, which correlates with the experiment performance mentioned above. So, a smaller compactness results in better performance. Moreover, when isolating just the insert algorithms, the exact method tends to have smaller compactness compared to approximate methods, which also correlates with the experimental results where exact inserts outperform approximate insert methods. But the exact and relaxed batch construction compactness measures do not correlate with the experiment performance results. So we used "overlap" to explain the CCT quality in this case.

To measure overlap, we count all nodes that overlap (cover) a given leaf trajectory. We refine this measure by comparing the depth of each leaf with the total number of cover-nodes and averaging over all leafs, but the key point is that it is simply measuring how much of the tree covers each leaf. Smaller overlap measures result in better query performance, and vice versa. Intuitively this method of measuring overlap makes sense, since data sets with higher intrinsic dimensionality contain trajectories that are harder to 'separate' from each other, which can result in higher overlap in a tree. If a leaf is covered by many nodes, then constructing and searching is harder since there are more potential nodes

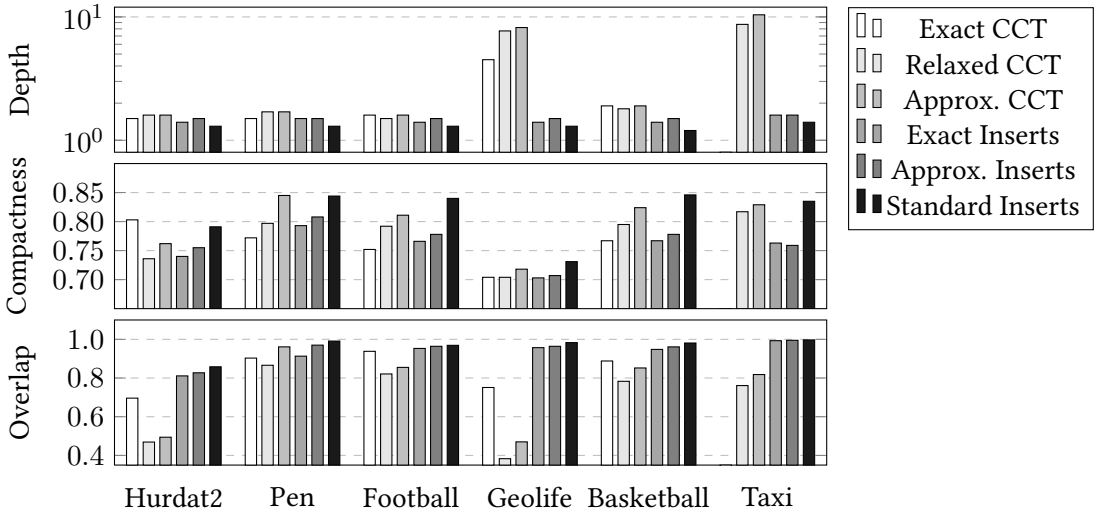

Figure 4: CCT Quality for batch construction and insertion algorithms on the six largest real data sets (c.f. Section 4.3). The average leaf depth (top) is normalized to an optimal depth ($\lceil \log_2 |\mathcal{S}| \rceil$). Compactness (middle) is the average ratio of child-parent radii, and overlap (bottom) is the average ratio of each trajectory's leaf depth and number of other node clusters that cover it. For the Taxi [67, 68] data set the Exact CCT batch construction did not finish within 3 days and is omitted.

to traverse. The batch construction algorithms tend to have smaller overlap than inserts, and exact algorithms have smaller overlap than their approximate counterparts (since approximate algorithms can result in larger radii).

One interesting and initially unexpected result in the experiments was that the Relaxed CCT outperformed the Exact CCT. The Relaxed CCT is constructed with fewer distance calls and essentially omits the trajectory reassigning component, compared to the Exact method, so we anticipated a trade-off at query time for the Relaxed method. However, the opposite occurred. The reason for this behavior is due to the overlap difference. The trajectory reassigning component of the Exact batch construction can lead to a larger overlap since trajectories can be reassigned multiple times during the iterations which can lead to more parent nodes that cover them.

Various data sets can also exhibit different quality measures depending on their intrinsic dimensionality. Figure 5 compares two real data set Relaxed CCT dendrograms. The Cats [45] data set has smaller intrinsic dimensionality compared to the Gulls [63] data set, and the dendrograms show this relationship with Cats having smaller compactness and overlap measures. Experiments (e.g. Figure 10) verify that the Cats Relaxed CCT outperforms the Gulls Relaxed CCT.

An attempt was made to measure the quality of the underlying data sets using the intrinsic dimensionality measure of [20]. Calculations showed that this measure was useful for data sets with normal distributions of pairwise distances, however, most real data sets in our study do not have this property and the measure did not accurately convey the underlying intrinsic dimensionality. In our setting, the overlap measure was a better indicator for the ease or difficulty of searching the data set.

4.4 Differences to Related Approaches

Multi-way metric indexes such as Cover-Trees [14] also provide the Nesting property, besides additional compactness and separation properties (Cover Trees use $1/1.3 \approx 0.78$ for compactness and separation in practice to balance arity and depth). Internal nodes of Cover-Trees have an assigned integer level and the distance between the center of a node with level i and the center of any of its descendants is no more than 2^i (c.f. Theorem 2 in [14]). Using these coarse values as radii, we have that every

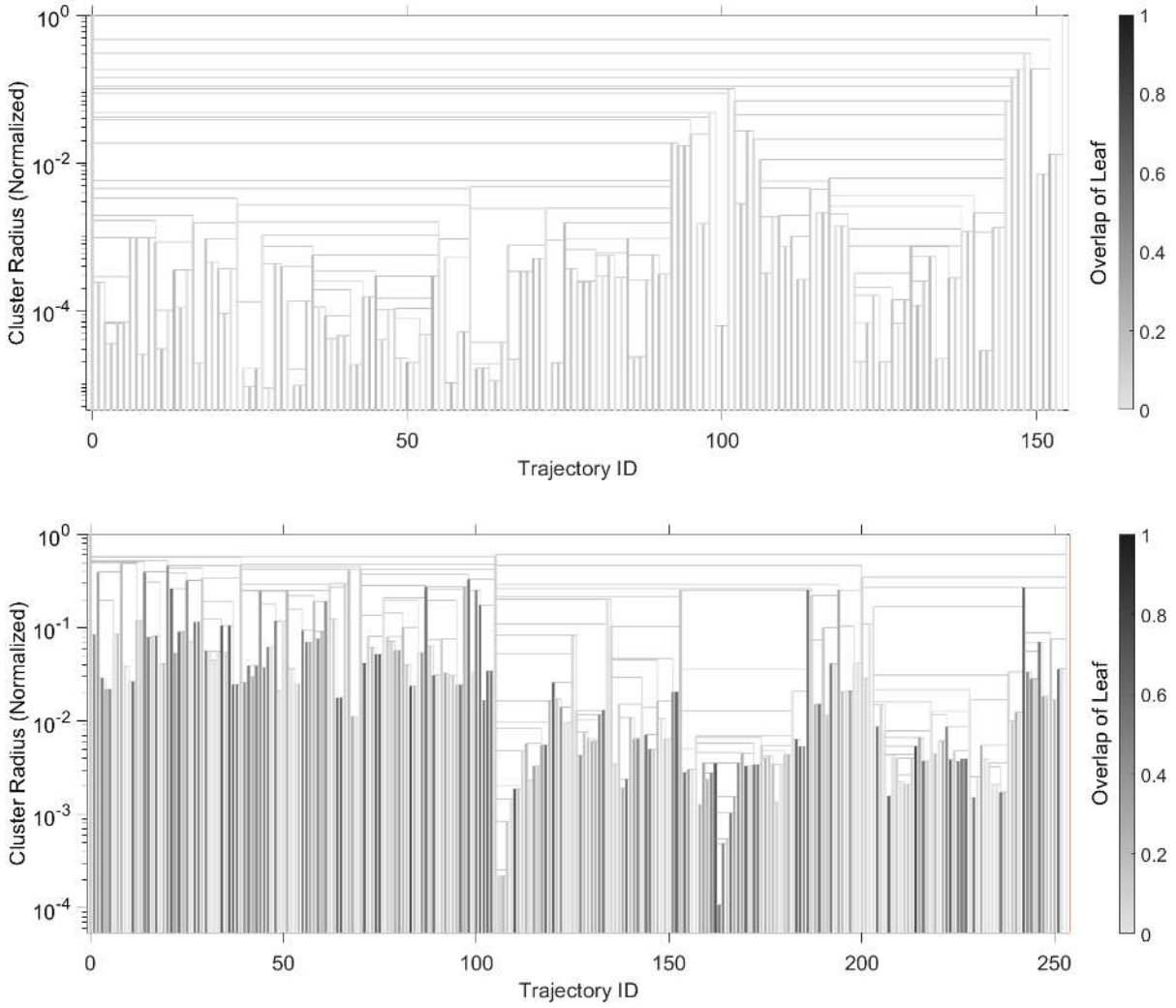

Figure 5: Relaxed CCT Dendrograms for Cats (top) and Gulls (bottom) real data sets (c.f. Section 4.3). The x-axis shows the number of input trajectories, y-axis the normalized cluster radii (compactness), and horizontal lines the parent nodes (tree depth). The vertical lines represent the leaf trajectories with lighter and darker shades corresponding to smaller and larger overlap measures, respectively. Tree balance is observed by the relative position of the vertical cut line beneath a parent that separates its two children.

Cover-Tree is a CCT. Their dynamic insertion and deletion of a single element performs no more than $\mathcal{O}(\gamma^6 \log |\mathcal{S}|)$ operations, which are mainly distance computations, where $2 \leq \gamma \leq |\mathcal{S}|$ denotes the expansion constant of the data set (c.f. Sections 1.1 and 2.2). For large trajectory data sets however, $\mathcal{O}(|\mathcal{S}|\gamma^6 \log |\mathcal{S}|)$ Fréchet distance computations might well be impractical, even for moderate γ values.

Though one may modify CCTs such that leafs store 'chunks' (fixed size subsets of trajectories) like practical implementations do (e.g. M-Trees [21]), this seems detrimental for the computationally expensive Fréchet distance in our setting.

It is important to note that the bound algorithms in Section 3 are independent of the CCT structure. This allows the flexibility to extend the query algorithms (c.f. Section 5) with further, e.g. data domain specific, heuristic bounds *without* the need to rebuild the data structure. This is in strong contrast to pruning approaches that use dD-Trees [11], Range-Trees [12], and grid-based hash structures, as in [19, 29, 26], for e.g. trajectories' start and end points in $\mathbb{R}^d$.

5 Proximity Queries

Our query algorithms for CCTs consists of three stages:

1. **Prune:** Collect candidate trajectories into a set $\mathcal{S}_1$ by performing a guided depth-first-traversal of the CCT, in which sub-tree's clusters may be excluded in a pre-order fashion using the triangle inequality, the cluster radius, and bound computations.

2. **Reduce:** Filter trajectories in $\mathcal{S}_1$ using heuristic proximity predicates and orderings of the approximate distance intervals to obtain a smaller set $\mathcal{S}_2$.

3. **Decide:** Finalize the result set by removing ambiguity in $\mathcal{S}_2$ that exceeds the specified query error, by potentially performing δ_F and/or δ_{FD} calls.

To gain some intuition regarding the effectiveness of this 3 stage approach, refer to Figure 6, which shows the Prune and Reduce stages for two kNN queries. The Prune stage generally searches a small subset of the CCT (by eliminating sub-trees) and returns a small candidate set. The Reduce stage can further exclude candidates, and also include candidates in the final result set. Distance calls are only employed in the Decide stage, by which time the number of remaining candidates are typically small (or often zero).

The following describes each query algorithm in the additive error model and the changes for the multiplicative error model are briefly noted in each section.

5.1 Approximate and Exact kNN Queries

Consider a query $k\text{NN}(Q, \varepsilon^+ \geq 0, k \geq 1)$ on $\mathcal{S}$, as defined in Section 2.1. We describe the three stages of our query algorithm.

1. Prune: Our query method heuristically guides the tree traversal towards a potentially close leaf. Recursively traverse the tree from the root, and for an internal node v, first descend to the child u that has the smallest lower bound $\text{LB}_\text{F}(Q, C(u))$ among the children of v. When a leaf is reached, append its trajectory to the initially empty set $\mathcal{S}_1$.

Once $|\mathcal{S}_1| \geq k$, prune sub-trees as follows. Track the kth smallest upper bound β_k in $\mathcal{S}_1$ using a heap, and only descend below node v if $\text{LB}_\text{F}(C(v), Q) \leq \beta_k + rad(v) - \varepsilon^+$. When a leaf node is reached, append its trajectory P to $\mathcal{S}_1$ only if $\text{LB}_\text{F}(P, Q) < \beta_k$ and either $\text{UB}_\text{F}(P, Q) < \beta_k$ or $\text{LB}_\text{FD}(P, Q, \beta_k) = false$.

2. Reduce: From $\mathcal{S}_1$, we filter with the final β_k value to obtain at least k elements in $\mathcal{S}_2$. That is, for those $P \in \mathcal{S}_1$ having $\text{UB}_\text{F}(P, Q) > \beta_k$, keep only those trajectories with $\text{LB}_\text{F}(P, Q) < \beta_k - \varepsilon^+$ and $\text{LB}_\text{FD}(P, Q, \beta_k - \varepsilon^+) = false$.

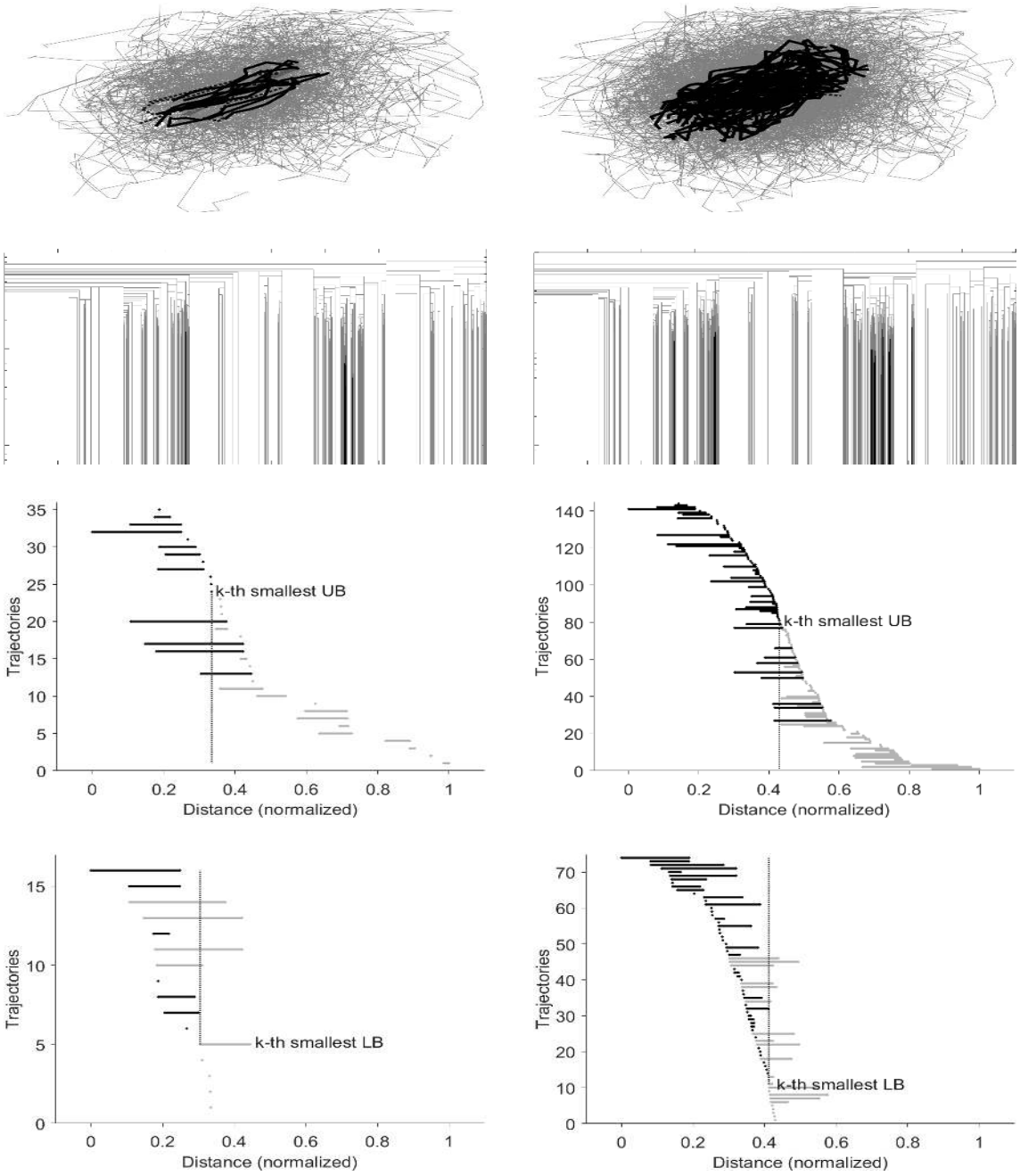

Figure 6: Two exact kNN queries for k=12 (first column) and k=64 (second column) on the Football [57] data set using the Relaxed CCT (c.f. Section 5). The first row shows 2D trajectory plots and the second row contains dendrograms that show the CCT prune stage search, both with the same legend as in Figure 1 (pruned trajectories are omitted). The third row shows trajectory bound intervals in $\mathcal{S}_2$, i.e. the upper/lower bound distances of a trajectory to the query. The trajectories in light grey show those that can be deleted in the reduce stage, since $\text{LB}_{\text{F}}(P,Q) + \varepsilon^+ > \beta_k$. The last row shows trajectory bound intervals in $\mathcal{S}_2$, including those that can be included (black) in result set $\mathcal{S}_{k\text{NN}}$ in the reduce stage, since $\text{UB}_{\text{F}}(P,Q) - \varepsilon^+ < \alpha_k$.

If $|\mathcal{S}_2| = k$, we are done and return the set $\mathcal{S}_{k\text{NN}} := \mathcal{S}_2$. Otherwise, locate the $(k+1)$-th smallest lower bound α_{k+1} in $\mathcal{S}_2$. For each $P \in \mathcal{S}_2$ with $\text{UB}_\text{F}(P,Q) - \varepsilon^+ < \alpha_{k+1}$ immediately move P from $\mathcal{S}_2$ to the initially empty set $\mathcal{S}_{k\text{NN}}$.

For the relative error model, first compute the kth smallest lower bound α_k in $\mathcal{S}_1$, set $\varepsilon^+ := \varepsilon^* \cdot \alpha_k$, and run stage two exactly as described above.

3. Decide: Perform the following until $|\mathcal{S}_{k\text{NN}}| = k$. Randomly choose a pivot trajectory $P \in \mathcal{S}_2$, compute $\pi := \delta_F(P,Q)$, and partition $\mathcal{S}_2$ by computing if the trajectory is closer or further from Q than π (use upper/lower bounds, and if it's undetermined compute the Fréchet decision procedure).

If the number of trajectories closer to Q than π is at most $k - |\mathcal{S}_{k\text{NN}}|$, append the closer trajectories to $\mathcal{S}_{k\text{NN}}$ and delete them from $\mathcal{S}_2$. Otherwise, delete the trajectories further from Q than π from $\mathcal{S}_2$.

Algorithm Analysis. Using a similar analysis as in the QuickSelect algorithm [31], the number of δ_F calls and δ_FD calls in the Decide stage is $\mathcal{O}(\log |\mathcal{S}_2|)$ expected and $\mathcal{O}(|\mathcal{S}_2|)$ expected, respectively. In the worst-case, no trajectories are discarded in the first two stages and $|\mathcal{S}_2| = |\mathcal{S}|$. However, experiments (c.f. Section 6.2.1) show much fewer distance computations than this worst-case analysis.

5.1.1 Optimization for NN Queries

We describe modifications for a NN algorithm that empirically performs slightly fewer distance computations than the kNN algorithm when $k = 1$ (c.f. Section 6.2.2).

1. Prune: We perform the following additional check when at a leaf node v: If $\text{UB}_\text{F}(C(v),Q) \leq \varepsilon^+$ is $true$ proceed to the next stage with $\mathcal{S}_1 := \{C(v)\}$.

2. Reduce: Same as kNN.

3. Decide: If $|\mathcal{S}_2| = 1$, we are done and return $\mathcal{S}_2$. Otherwise, compute the second-smallest lower bound α_2 in $\mathcal{S}_2$, with associated trajectory P. If $\text{LB}_\text{FD}(P,Q,\alpha_2) = false$ but $\delta_\text{FD}(P,Q,\alpha_2) = true$ then return $\{P\}$.

Otherwise, sort $\mathcal{S}_2$ ascending by the upper bound, and loop on each $P \in \mathcal{S}_2$ to track the current best trajectory P' and its distance $\pi := \delta_F(P',Q)$. For subsequent $P \in \mathcal{S}_2$, if $\text{LB}_\text{FD}(P,Q,\pi) = false$ but $\delta_\text{FD}(P,Q,\pi) = true$, then set $P' := P$ and $\pi := \delta_F(P,Q)$. Finally return $\{P'\}$.

5.2 Approximate and Exact RNN Queries

Consider a range query $\text{RNN}(Q, \tau \geq 0, \varepsilon^+ \geq 0)$ on $\mathcal{S}$, as defined in Section 2.1. For the queries under the relative error model, we set $\varepsilon^+ := \varepsilon^* \cdot \tau$.

1. Prune: Recursively traverse the tree from the root. For an internal node v, only descend to its children if $\text{LB}_\text{F}(C(v),Q) \leq \tau + rad(v)$. That is, the associated cluster of v may contain trajectories within distance τ of Q. When a leaf is reached, append its stored trajectory P to the initially empty set $\mathcal{S}_1$ if $\text{LB}_\text{F}(P,Q) \leq \tau$.

All trajectories within the cluster of a node v may immediately belong in the result set $\mathcal{S}_\text{RNN}$, so we can potentially finish the sub-tree of v with a UB_F call. Since our UB_F call is more expensive than LB_F calls, we speed up the search using a heuristic parameter[2] $\kappa \geq 1$ in the following: Only if $\kappa \cdot \text{LB}_\text{F}(C(v),Q) + rad(v) < \tau$ check $\text{UB}_\text{F}(C(c),Q) + rad(c) \leq \tau$ and, on success, simply append all leafs beneath v to the initially empty set $\mathcal{S}_\text{RNN}$.

2. Reduce: For each trajectory $P \in \mathcal{S}_1$, if $\text{UB}_\text{F}(P,Q) < \tau + \varepsilon^+$, then append P to $\mathcal{S}_\text{RNN}$, else if $\text{LB}_\text{FD}(P,Q,\tau) = false$ then append P to initially empty set $\mathcal{S}_2$, otherwise P is discarded.

3. Decide: For each trajectory $P \in \mathcal{S}_2$, if $\delta_\text{FD}(P,Q,\tau) = true$, then append P to $\mathcal{S}_\text{RNN}$.

Algorithm Analysis. In the worst case no trajectories are discarded in the first two stages, hence, the query algorithm might perform $\mathcal{O}(|\mathcal{S}|)$ bound computations in the Prune and Reduce stages, and $\mathcal{O}(|\mathcal{S}|)$ Fréchet decision procedure computations in the Decide stage.

[2]Our experiments use $\kappa = 1.25$, since this matches the average upper/lower bound ratio we observe on elements of the data sets.

However, our experiments in Section 6.2.1 (see Figure 12) show much fewer bound computations and δ_{FD} calls.

5.3 Implicit Approximate Queries

We also describe a variant of kNN and RNN query algorithms that perform no distance and no Fréchet decision procedure computations. Instead, implicit approximation query algorithms return trajectory results with the smallest additive ε^+ or relative ε^* approximation error, which is part of the output. Since results are determined by the set of heuristic bounds, this method can result in a significant computational speed-up over aforementioned query algorithms.

The Prune and Reduce stages of the implicit approximate RNN and kNN query algorithms are the same as their counterparts above with $\varepsilon^+ := 0$. The modified Decide stages are as follows.

kNN **Decide:** If $|\mathcal{S}_2| = k$, then set $\mathcal{S}_{k\text{NN}} := \mathcal{S}_2$. Otherwise, sort $\mathcal{S}_2$ by upper bound ascending, and set $\mathcal{S}_{k\text{NN}}$ to the first k elements in $\mathcal{S}_2$.

To compute ε^+ and ε^*, set β_k to the k-th smallest upper bound in $\mathcal{S}_2$. Delete the first k elements in $\mathcal{S}_2$, sort $\mathcal{S}_2$ by lower bound ascending, and set α_k to the lower bound of the first element in $\mathcal{S}_2$. Set $\varepsilon^+ := \beta_k - \alpha_k$. Set $\varepsilon^* := (\beta_k - \alpha_k)/\alpha_k$.

RNN **Decide:** Set $\mathcal{S}_{k\text{NN}} := \mathcal{S}_2$.

To compute ε^+ and ε^*, set β_k to the largest upper bound in $\mathcal{S}_2$. Set $\varepsilon^+ := \beta_k - \tau$ and $\varepsilon^* := (\beta_k - \tau)/\tau$.

6 Experiments

We experimentally evaluate the scalability, effectiveness and efficiency of bounds in Section 3, data structure constructions in Section 4, and query algorithms in Section 5. As introduced in Section 1, our measurements focus on the primary empirical goal of measuring the number of distance computations, with a subordinate goal of measuring the query I/O (tree node accesses).

We compare our contribution to several competitors, including a recent state-of-the-art contribution [16] for RNN queries among 2D trajectories (which improves upon previous RNN search approaches on 2D data [10, 19, 29]), a standard M-Tree [21], a standard Cover-Tree [14], and an improved linear scan algorithm (Section 6.1.3). Although the approach [26] is most similar in regard of the supported operations, it does not allow practical comparison on our test data sets due to its exponential construction time and data structure size.

6.1 Experiment Setup

We now describe how the experiments are setup whereas Section 6.2 discusses the results[3].

6.1.1 Real Data Sets

We obtained sixteen real-world data sets [23, 32, 34, 35, 45, 50, 52, 54, 55, 56, 57, 63, 64, 67, 68] of diverse origin and characteristics to evaluate our data structure construction and query algorithms (see Table 3). To broaden our experiments, but also to challenge our bound algorithms, we use the trajectory simplification algorithm of [3] to obtain trajectories whose sampling are irregular (c.f. Section 6.1.1). Given an error bound $\widehat{\varepsilon} \geq 0$, this simplification algorithm returns a trajectory over a subset of the original vertices whose Fréchet distance is within the specified bound. For every $P \in \mathcal{S}$, we set $\widehat{\varepsilon}$ to be a small percentage (typically 1% or 2%) of $reach(P)$, where $reach$ denotes as the maximum distance from a trajectory's start vertex to any of its other vertices (see e.g. [26]). We found that this substantially reduces the time required to run the experiments, without materially changing the results.

[3] See https://github.com/japfeifer/frechet-queries for more detailed experimental results, the code, and the data sets.

Data Set	$\lvert\mathcal{S}\rvert$	d	Vertices orig.	Vertices simpl.	Trajectory Description
Vessel-M [50]	106	2	23.0	7.3	Mississippi river shipping vessels Shipboard AIS.
Pigeon [34]	131	2	970.0	26.0	Homing Pigeons (release sites to home site).
Seabird [55]	134	2	3175.8	43.5	GPS of Masked Boobies in Gulf of Mexico.
Bus [32]	148	2	446.6	40.3	GPS of School buses.
Cats [45]	154	2	526.1	34.2	Pet house cats GPS in Raleigh-Durham, NC, USA.
Buffalo [23]	165	2	161.3	54.5	Radio-collared Kruger Buffalo, South Africa.
Vessel-Y [50]	187	2	155.2	4.0	Yangtze river shipping Vessels Shipboard AIS.
Gulls [63]	253	2	602.1	33.7	Black-backed gulls GPS (Finland to Africa).
Truck [32]	276	2	406.5	41.4	GPS of 50 concrete trucks in Athens, Greece.
Bats [35]	545	2	44.1	7.3	Video-grammetry of Daubenton trawling bats.
Hurdat2 [54]	1788	2	27.7	7.9	Atlantic tropical cyclone and sub-cyclone paths.
Pen [64]	2858	2	119.8	24.4	Pen tip characters on a WACOM tablet.
Football [57]	18034	2	203.4	15.4	European football player ball-possession.
Geolife [52]	18670	2	1332.5	14.2	People movement, mostly in Beijing, China.
Basketball [56]	20780	3	44.1	7.3	NBA basketball three-point shots-on-net.
Taxi [67, 68]	180736	2	343.0	41.3	10,357 Partitioned Beijing taxi trajectories.

Table 3: Real data sets, showing number of input trajectories $-\mathcal{S}-$, dimensions d, average number of original vertices per trajectory, average number of simplified vertices per trajectory, and a description.

Though some of these real data sets have a small number of trajectories (e.g. Vessel-Y vs. Taxi), they are included in our experiments since they show that proximity queries in small sets can cause more distance calls than searches in larger sets (e.g. Figures 8, 15, 16, and 18).

We use two methods to generate query trajectories for the real data sets. Method one randomly selects an input trajectory P, perturbs its vertices up to 3% and translates it up to 5% of $reach(P)$ uniformly at random. For direct comparison, method two uses the query generator of [16], that returns exactly $10, 100$ or 1000 results for a RNN query. We generated 1000 query trajectories per data set with either method. Results based on the second query generation method indicate that in the respective figure.

6.1.2 Synthetic Data Sets

Testing on synthetic data sets helps to analyze which characteristics most impact the number of δ_F calls and overall query efficiency. By varying a single characteristic while holding others constant, the impact of the particular characteristic on the measurements can be assessed. The routine to create these data sets is parameterized by the following characteristics:

- cluster size α_{CS} (number of trajectories per cluster),
- trajectory straightness factor α_{SF} and maximum edge distance α_{ED},
- average trajectory size n,
- number of trajectories $|\mathcal{S}|$, and
- spatial dimensions d.

Our baseline synthetic data set is generated with the values $\alpha_{CS} = 10$, $\alpha_{SF} = 0.95$ with $\alpha_{ED} = 0.6$, $n = 15$, $|\mathcal{S}| = 5000$, and $d = 2$. For the experiments, we vary $\alpha_{CS} \in \{1, 10, 25, 50, 100\}$,

$\alpha_{SF} \in \{0.5, 0.8, 0.9, 0.95, 0.99\}$, $n \in \{15, 25, 35, 45, 55\}$, $d \in \{2, 4, 8, 16, 32\}$, and the number of trajectories $|\mathcal{S}|$ in $\{5\text{K}, 10\text{K}, 20\text{K}, 30\text{K}, 40\text{K}, 1\text{M}, 10\text{M}\}$.

Synthetic data sets and their associated query trajectories are created in the following four steps.

Step 1: Unique (non-clustered) trajectories. First, increase the designated number of trajectories $|\mathcal{S}|$ by 500. Generate each of the $|\mathcal{S}|/\alpha_{CS}$ trajectories with the following random-walk routine. Choose a number of vertices $z \in \left[\frac{n}{2}, \frac{3n}{2}\right]$ uniformly at random and then choose the initial vertex $p_1 \in [0,1]^d$ uniformly at random. Subsequent vertices p_i are created with

$$p_i := \alpha_{ED} \cdot \sigma + p_{i-1} + \alpha_{SF} \cdot (p_{i-1} - p_{i-2}) \quad ,$$

where each random step $\sigma \in [0,1]^d$ is chosen uniformly.

Step 2: Clustered trajectories. For each unique trajectory generate a copy of it, perturb uniformly at random the copy's vertices up to the maximum edge distance α_{ED}, and then translate uniformly at random the copy up to the maximum edge distance. This process is performed $\alpha_{CS} - 1$ times per unique trajectory.

Step 3: Sample query trajectories. Out of the above set $\mathcal{S}$, we choose 1000 trajectories uniformly at random without replacement.

Step 4: Add 'noisy' trajectories. Finally, 500 additional 'noise' trajectories are generated as in Step 1.

6.1.3 Improved NN Linear Scan

Given the lack of available algorithms for exact nearest-neighbor search under the Fréchet distance and our discussion on the 'curse of dimensionality' (c.f. Section 1), we implemented a competitor, called improved NN linear scan, suitable for high dimensional trajectory data.

The improved NN linear scan algorithm leverages our bounds of Section 3 by checking each $P \in \mathcal{S}$, and appending P to the initially empty set $\mathcal{S}_1$ if $\text{LB}_{\text{F}}(P, Q) < \beta$ and $\text{LB}_{\text{FD}}(P, Q, \beta) = \mathit{false}$. The smallest upper bound β is tracked, upper bound $\text{UB}_{\text{F}}(P, Q)$ is only computed when P is appended to $\mathcal{S}_1$, and $\text{LB}_{\text{FD}}(P, Q, \beta)$ is only computed when $\text{LB}_{\text{F}}(P, Q) < \beta$.

6.1.4 Quality of the Data Structure

6.2 Experimental Results

Note that the results on the quality of the CCT data structure are in Section 4.3. Experimental results are separated into primary results, which evaluate the proposed Relaxed CCT method on real and synthetic data sets and compare it with related work, and supplementary results, which compare the different exact and approximate variations of our approaches against each other.

6.2.1 Primary Results

Figures 7 and 8 show the effectiveness of exact NN queries on Relaxed CCTs for synthetic and real data sets, respectively. On most data sets, the average number of expensive δ_F distance calls per query is *one or fewer*, and only increases slightly for highly clustered data sets. Surprisingly, the majority of queries require no distance computations at all for many of the data sets. The 10M trajectory data set performs on average only 1.04 expensive δ_F calls per query. Interestingly, the Vessel-Y [50] data set requires a similar average of 0.97 δ_F calls, even though it is a much smaller data set. The Vessel-Y data set has higher intrinsic dimensionality, so this shows that clustering of data has a much larger influence on distance calls than the number of trajectories does. The number of node visits (normalized to a factor of $|\mathcal{S}|$) decreases as the number of trajectories increases, showing effective pruning of the search space.

Figures 9 and 10 show the effectiveness of exact kNN queries on Relaxed CCTs for synthetic and real data sets, respectively. The results correspond to the NN query results above.

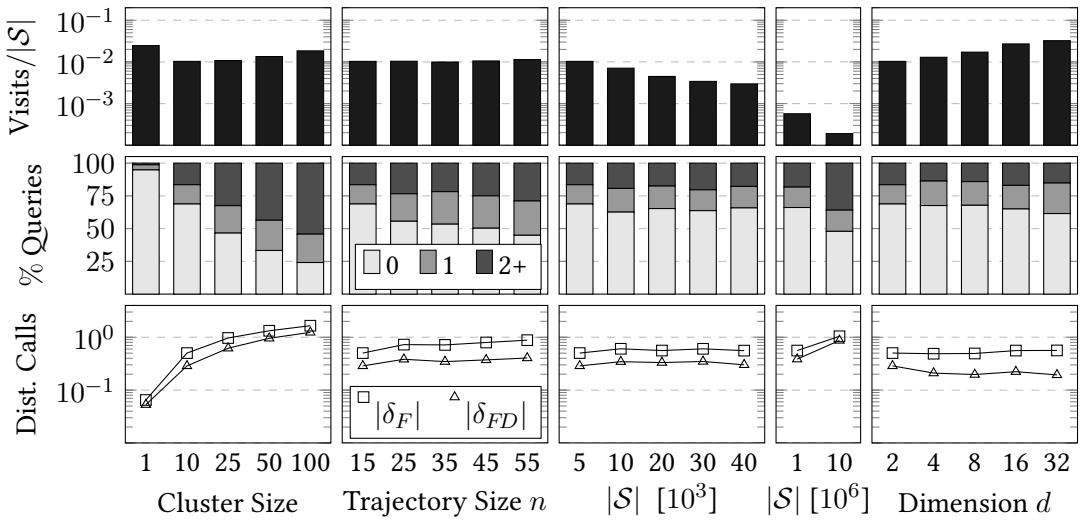

Figure 7: Effectiveness of exact NN queries on synthetic data set Relaxed CCTs, averaged over 1000 queries (c.f. Section 6.2.1). The top row shows average number of tree node visits (normalized to a factor of $|\mathcal{S}|$). The middle row shows the percentage of queries that performed 0, 1, or more than 1 distance computation. The bottom row shows the *absolute number* (not normalized) of δ_F and δ_{FD} calls.

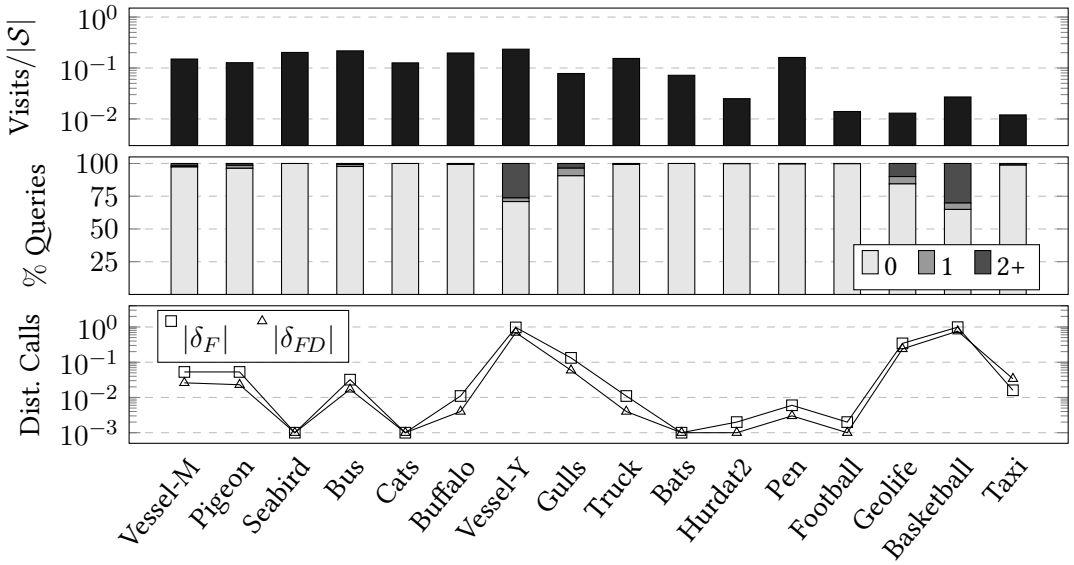

Figure 8: Effectiveness of exact NN queries on real data set Relaxed CCTs, averaged over 1000 queries (c.f. Section 6.2.1). The top row shows average number of tree node visits (normalized to a factor of $|\mathcal{S}|$). The middle row shows the percentage of queries that performed 0, 1, or more than 1 distance computation. The bottom row shows the *absolute number* (not normalized) of δ_F and δ_{FD} calls.

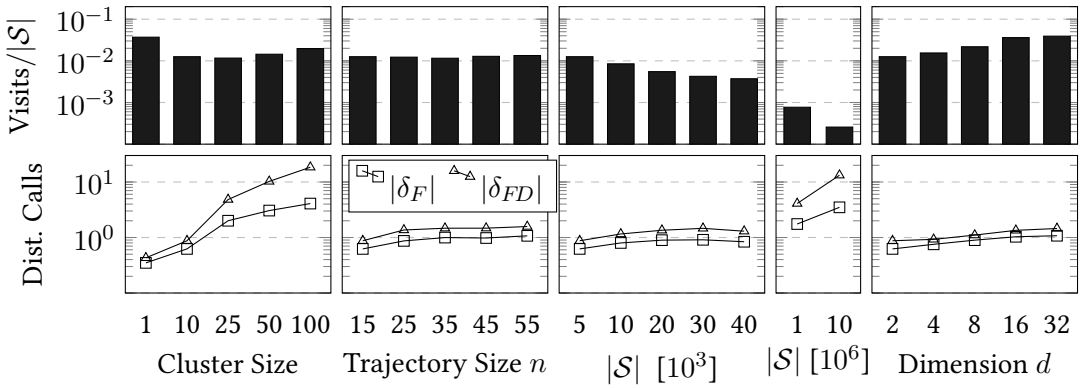

Figure 9: Effectiveness of exact kNN queries ($k = 5$) on synthetic data set Relaxed CCTs, averaged over 1000 queries (c.f. Section 6.2.1). The top row shows average number of tree node visits (normalized to a factor of $|\mathcal{S}|$). The bottom row shows the *absolute number* (not normalized) of δ_{FD} and δ_F calls.

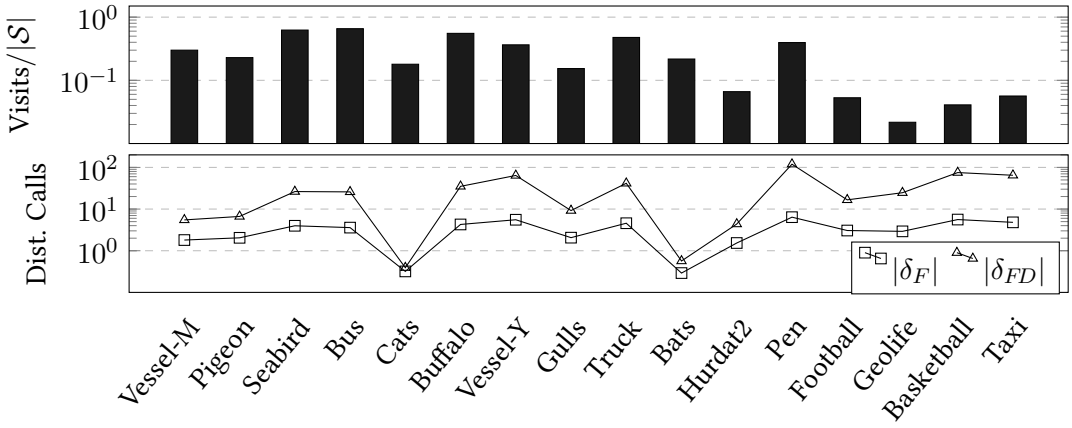

Figure 10: Effectiveness of exact kNN queries ($k = 5$) on real data set Relaxed CCTs, averaged over 1000 queries (c.f. Section 6.2.1). The top row shows average number of tree node visits (normalized to a factor of $|\mathcal{S}|$). The bottom row shows the *absolute number* (not normalized) of δ_{FD} and δ_F calls.

The experimental results for comparison of the Relaxed CCT vs. standard, 'off-the-shelf' metric indexing methods M-Tree [21] and Cover-Tree [14] are in Figure 11. The Fréchet distance function is 'plugged' into the generic Cover-Tree, whose implementation uses a 'scaling' constant of 1.3 which results in $1/1.3 \approx 0.78$ for compactness and separation to balance arity and depth. For the M-Tree, we used the random promote method, as it performs the fewest distance calls during construction, and set the maximum arity to 100. We also attempted to improve M-Tree performance by first testing δ_{FD}, and if it fails then calling δ_F, for both construction and queries. The results show that both for construction and query the number of δ_F calls for the CCT are usually at least an order of magnitude smaller than required for the standard M-Tree and Cover-Tree. For example, the kNN queries on the Taxi [67, 68] data set performed 6.0 δ_F calls on average using the CCT, and 16.4×10^3 calls using the Cover-Tree.

Figure 12 compares the performance of our approach with those of the recent contribution by Bringmann et al. [16] that performs exact RNN queries under the Fréchet distance in 2 dimensional space, using an 8 dimensional KD-tree (c.f. Section 1). For the KD-Tree based approach, the number of visits is defined as the total number of nodes visited during the tree traversal. In the bound invocation metric, four bounds (LB_{FD},UB_{ADF1},UB_{ADF2},UB_{ADF3}) may be counted for the CCT and only three (adaptive equal-time, negative filter, and greedy) for [16]. In comparison, the RNN queries using CCTs have fewer node visits, compute fewer bound computations, and perform fewer Fréchet decision calls by

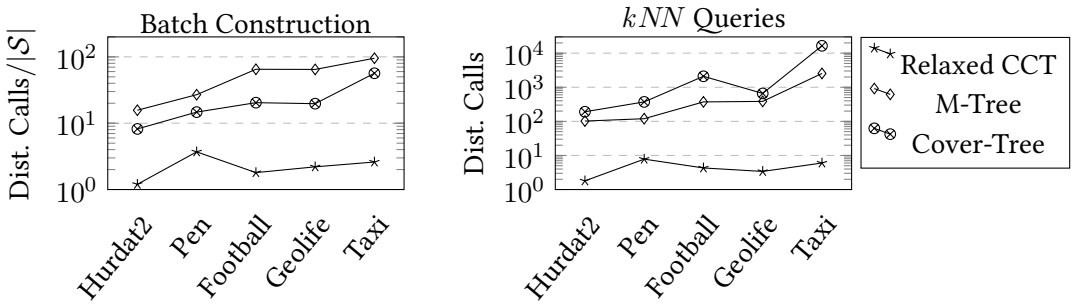

Figure 11: Performance of Relaxed CCTs vs. standard M-Tree [21] and Cover-Tree [14] implementations for the five largest real $d = 2$ data sets (c.f. Section 6.2.1). The left chart shows construction δ_F calls normalized over the data set size $|\mathcal{S}|$. The right chart shows exact kNN query ($k = 10$) *absolute number* (not normalized) δ_F calls, averaged over 1000 queries (query method two).

an average factor of 3 for synthetic data sets. Though our queries may perform up to four bound computations per trajectory, and not just three, it is surprising that CCTs perform fewer total bound computations for all but one of the inputs. This improvement is due to stronger bounds and clustering of trajectories, which allows the algorithm to test if all trajectories within a cluster belong in the result.

Figure 13 compares the Prune stage of our NN query to the improved NN linear scan. Linear scan visits are defined as total trajectories scanned. With exception of the Pen [64] data set, the number of CCT visits are factors between ten to over one hundred times smaller than the linear scan's, and the number of CCT bound computations are ten times smaller than the linear scan's, especially for datasets with a large number of trajectories. Even in higher dimensions (e.g. $d = 32$), the CCT performs a factor of thirty fewer visits.

Figure 14 results show that the number of δ_F calls for the six types of CCT constructions, and corresponding node visits for NN queries. For CCT construction methods that perform δ_F calls, the Relaxed CCT performs the fewest, even sub-linear on Hurdat2, hence significantly fewer than $\mathcal{O}(|\mathcal{S}|^2)$. Note that the Exact CCT batch construction for the Taxi data set did not complete in a reasonable time due to the quadratic nature of the algorithm. We attempted to speed-up the Exact CCT batch construction algorithm by quickly eliminating trajectories outside of a 'neighborhood', but this improvement became less effective as $|\mathcal{S}|$ grew. The Relaxed CCT does not have this issue, and also shows the best query performance.

The node visits for all CCT constructions correlate with the overlap quality measure (see Section 4.3, Figure 4). The Relaxed CCT performs the fewest NN node visits at query time. Interestingly, the Approximate CCT has relatively good query performance, and can be useful in practice since its construction is faster than the Relaxed CCT since no δ_F calls are performed. The insert algorithms typically result in more query node accesses compared with batch constructions. The standard insert algorithm usually performs the worst at query time, especially if the data set has higher intrinsic dimensionality.

6.2.2 Supplementary Results

Figure 15 shows the gain in effectiveness from approximate over exact kNN queries, with $k = 5$ and $\varepsilon^* = 0.5$, on our real-world data sets. For the majority of the approximate queries, the number of δ_F and δ_{FD} calls are a factor of two or more smaller than those of exact queries. For the Pen [64] data set, the number of distance calls in an approximate query decreases by a factor of *forty*, suggesting that small approximation factors can result in significant performance gains.

Our new and improved bounds in Section 3 result in better query performance, as shown in Figure 16. For example, without the bound enhancements (using only previously existing bounds), the RNN

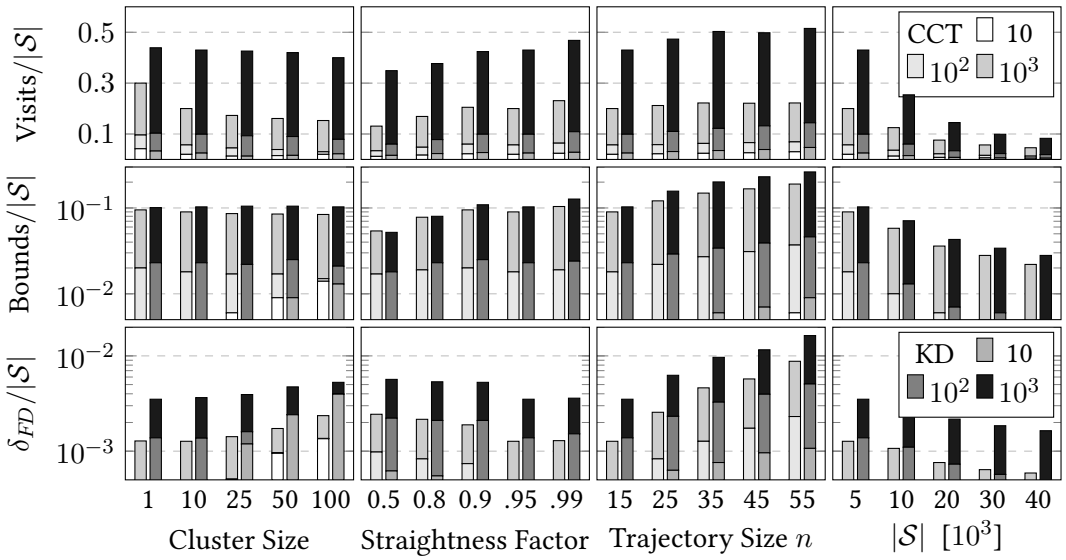

Figure 12: Performance of Relaxed CCTs vs. KD-Trees [16] for exact RNN-queries on synthetic data (c.f. Section 6.2.1). Bar charts show average metrics over 1000 RNN queries (query method two), chosen to return exactly 10, 100 or 1000 results, for CCTs (lighter shades) and KD-Trees (darker shades). All metrics are normalized over the data set size $|\mathcal{S}|$. The rows denote number of tree node visits (top) during the pruning stage, number of bound computations (middle), and the number of Fréchet decision procedure computations (bottom) in the last stage (c.f. Section 5.2).

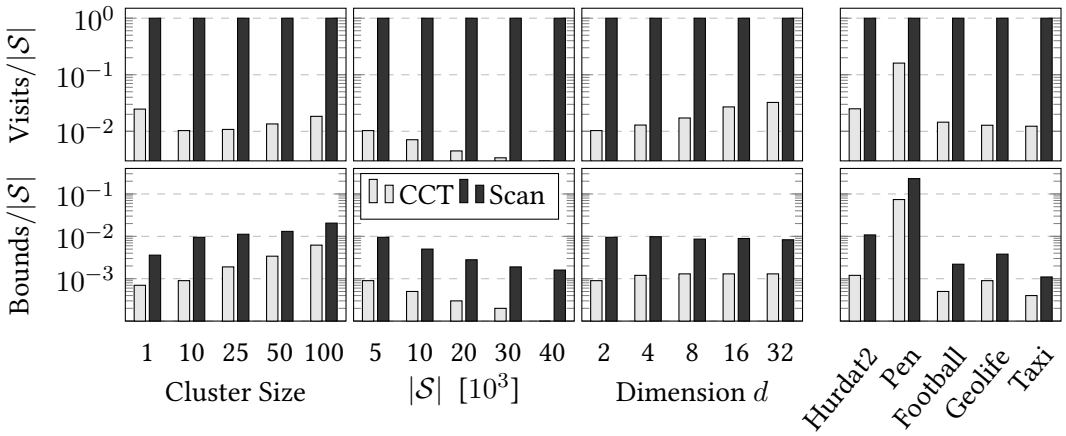

Figure 13: Performance of the Relaxed CCT Prune stage vs. Improved Linear Scan for exact NN-queries on synthetic (first 3 columns) and real (last column) data (c.f. Section 6.2.1). Bar charts show average metrics over 1000 queries for CCTs (light shade) and the improved Scan (dark shade). All metrics are normalized over the data set size $|\mathcal{S}|$. The rows denote number of node visits (top) during the pruning stage, and number of bound computations (bottom) in the reduce stage.

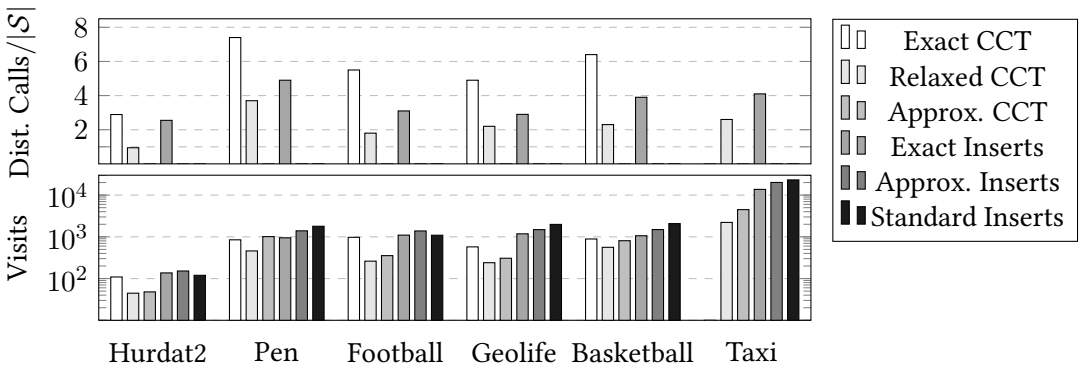

Figure 14: Effectiveness of CCT constructions and index query performance on the six largest real data sets (c.f. Section 6.2.1). The top row shows δ_F calls of batch constructions and dynamic insertions, normalized over the data set size $|\mathcal{S}|$. The bottom row shows tree node visits of exact NN queries, averaged over 1000 queries. For the Taxi [67, 68] data set the Exact CCT batch construction did not finish within 3 days and is omitted.

queries perform a factor of 4.7 more δ_{FD} calls on average for the five largest $d = 2$ real data sets.

Figure 17 shows that implicit approximate queries return, on average, results with small ε^* errors. All real data sets show $\varepsilon^* < 0.5$ for NN queries, and $\varepsilon^* < 1.8$ for kNN queries. Lower intrinsic dimensionality correlates with smaller ε^*, and vice versa.

In Section 5.1.1 we state that our optimized NN algorithm can outperform the kNN when $k = 1$, and results in Figure 18 provide evidence for the claim. For example, the NN query on the Basketball [56] data set performs a factor of two fewer δ_F calls and a factor of ten fewer δ_{FD} calls.

7 Directions for Future Work

Our experiments show that even slightly larger cluster radii can negatively impact metric pruning efficiency. We are therefore interested in other practical batch construction variants using Gonzalez' algorithm [36], or more recent techniques such as CLIQUE [5], SUBCLU [42], genetic algorithm clustering [9], mutual information hierarchical clustering [48], or belief propagation clustering [33].

The proposed 'Fix-Ancestor-Radius' primitive, which enables dynamic insertions, also allows to rectify radii that are affected from trajectory deletions in CCTs. We are interested in experiments on CCT quality and query performance in the fully-dynamic setting including identifying index sub-trees that benefit from a rebuild. It is also worthwhile exploring changes required to implement CCT algorithms on multi-way trees such as the M-tree [21], due to it's practical disk-based properties. It may also be interesting to extend this work to other trajectory distance metrics such as the Hausdorff [6], discrete Fréchet [17], and Wasserstein [60] distances, depending on application-specific requirements.

The kNN query algorithm analysis and experiment results show that the decide stage can perform $\mathcal{O}(|\mathcal{S}_2|)$ Fréchet decision procedure computations. Techniques, such as heuristic-guided pivot selection, may further reduce the number of δ_{FD} calls.

Finally, our future work seeks to investigate changes required to support proximity searches on sub-trajectories [25]. Algorithm modifications would need to balance cluster tree construction time, space consumption, and query time.

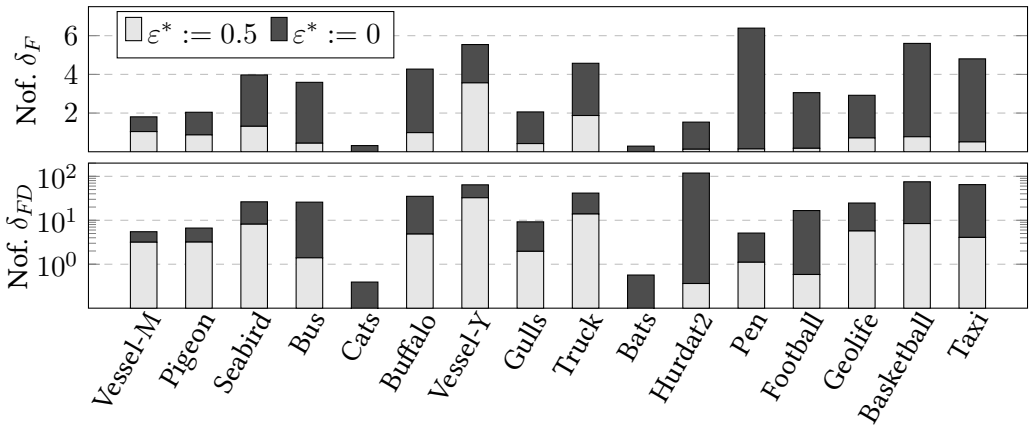

Figure 15: Effectiveness of exact vs. approximate ($\varepsilon^* = 0.5$) kNN queries on real data sets, for $k = 5$ (c.f. Section 6.2.2). Bar charts show average absolute values over 1000 queries for approximate (dark gray) and exact (light grey) queries, on Relaxed CCTs. The rows denote the number of distance δ_F (top) and Fréchet decision procedure δ_{FD} (bottom) computations during the Decide stage.

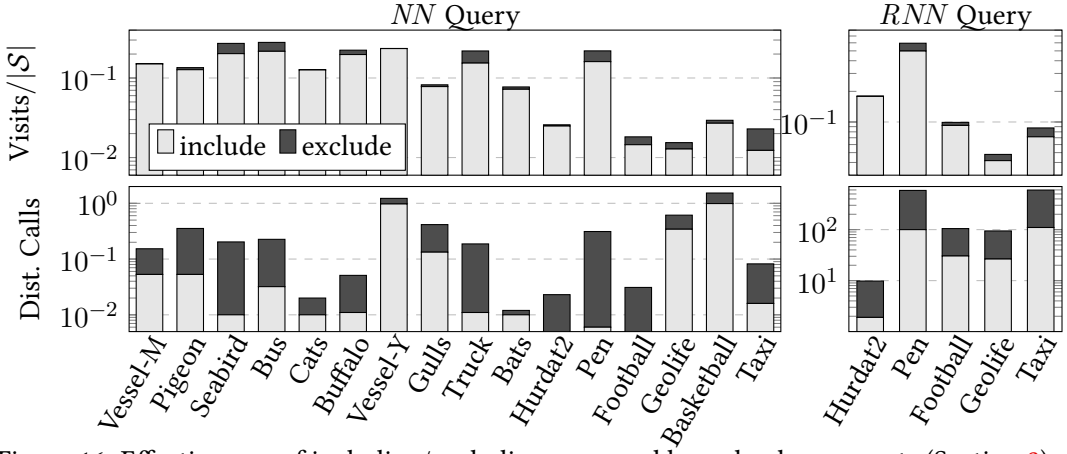

Figure 16: Effectiveness of including/excluding proposed bound enhancements (Section 3), on real data set Relaxed CCTs, averaged over 1000 queries (c.f. Section 6.2.2). The left side shows exact NN queries, and the right side shows exact RNN queries chosen to return exactly 100 results. The top row shows average number of tree node visits (normalized to a factor of $|\mathcal{S}|$). The bottom row shows the *absolute* δ_F and δ_{FD} calls for NN and RNN queries, respectively.

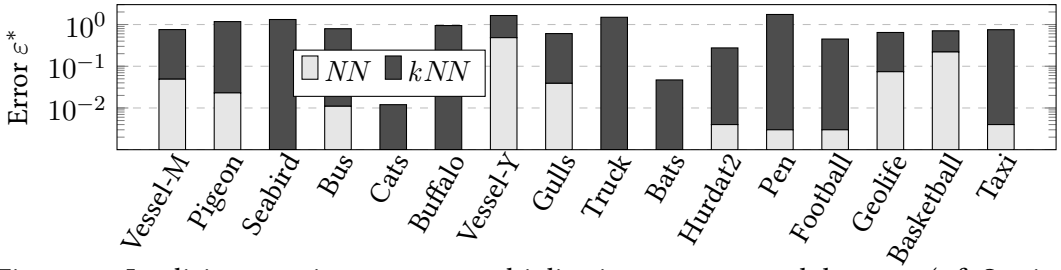

Figure 17: Implicit approximate query multiplicative errors on real data sets (c.f. Section 6.2.2). Bar chart shows average worst-case ε^* values over 1000 queries for NN (light shade) and kNN $k = 5$ (dark shade) queries, on Frugal CCTs.

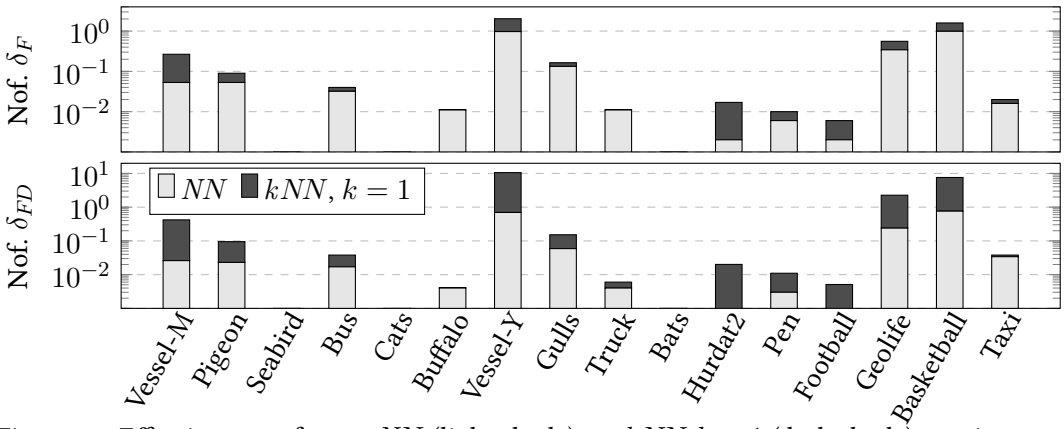

Figure 18: Effectiveness of exact NN (light shade) vs. kNN $k = 1$ (dark shade) queries on real data sets (c.f. Section 6.2.2). Bar charts show average absolute values over 1000 queries, on Relaxed CCTs. The rows denote the number of distance δ_F (top) and Fréchet decision procedure δ_{FD} (bottom) computations during the Decide stage.

References

[1] ACM. ACM SIGSPATIAL cup 2017 - range queries in very large databases of trajectories. http://sigspatial2017.sigspatial.org/giscup2017/, 2017.

[2] Agarwal, P. K., Avraham, R. B., Kaplan, H., and Sharir, M. Computing the discrete Fréchet distance in subquadratic time. *SIAM Journal on Computing 43*, 2 (2014), 429–449.

[3] Agarwal, P. K., Har-Peled, S., Mustafa, N. H., and Wang, Y. Near-linear time approximation algorithms for curve simplification. *Algorithmica 42*, 3-4 (2005), 203–219.

[4] Aggarwal, A., Vitter, J., et al. The input/output complexity of sorting and related problems. *Communications of the ACM 31*, 9 (1988), 1116–1127.

[5] Agrawal, R., Gehrke, J., Gunopulos, D., and Raghavan, P. Automatic subspace clustering of high dimensional data. *Data Mining and Knowledge Discovery 11*, 1 (2005), 5–33.

[6] Alt, H. The computational geometry of comparing shapes. In *Efficient Algorithms*. Springer, 2009, pp. 235–248.

[7] Alt, H., and Godau, M. Computing the Fréchet distance between two polygonal curves. *IJCGA 5*, 01n02 (1995), 75–91.

[8] Astefanoaei, M., Cesaretti, P., Katsikouli, P., Goswami, M., and Sarkar, R. Multi-resolution sketches and locality sensitive hashing for fast trajectory processing. In *Proceedings of the 26th ACM SIGSPATIAL Conference* (2018), ACM, pp. 279–288.

[9] Auffarth, B. Clustering by a genetic algorithm with biased mutation operator. In *IEEE Congress on Evolutionary Computation* (2010), IEEE, pp. 1–8.

[10] Baldus, J., and Bringmann, K. A fast implementation of near neighbors queries for Fréchet distance (GIS Cup). In *Proceedings of the 25th ACM SIGSPATIAL Conference* (2017), ACM, p. 99.

[11] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975), 509–517.

[12] Bentley, J. L. Decomposable searching problems. *Inf. Process. Lett. 8*, 5 (1979), 244–251.

[13] Bermingham, L., and Lee, I. A framework of spatio-temporal trajectory simplification methods. *International Journal of Geographical Information Science 31*, 6 (2017), 1128–1153.

[14] Beygelzimer, A., Kakade, S., and Langford, J. Cover trees for nearest neighbor. In *Machine Learning, Proceedings of the 23rd International ICML Conference* (2006), pp. 97–104.

[15] Bringmann, K. Why walking the dog takes time: Fréchet distance has no strongly subquadratic algorithms unless seth fails. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on* (2014), IEEE, pp. 661–670.

[16] Bringmann, K., Künnemann, M., and Nusser, A. Walking the dog fast in practice: Algorithm engineering of the fréchet distance. In *35th International Symposium on Computational Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA.* (2019), pp. 17:1–17:21.

[17] Bringmann, K., and Mulzer, W. Approximability of the discrete Fréchet distance. *Journal of Computational Geometry 7*, 2 (2015), 46–76.

[18] Buchin, K., Buchin, M., Meulemans, W., and Mulzer, W. Four soviets walk the dog: improved bounds for computing the Fréchet distance. *Discrete & Computational Geometry 58*, 1 (2017), 180–216.

[19] Buchin, K., Diez, Y., van Diggelen, T., and Meulemans, W. Efficient trajectory queries under the Fréchet distance (GIS Cup). In *Proceedings of the 25th ACM SIGSPATIAL Conference* (2017), ACM, p. 101.

[20] Chávez, E., Navarro, G., Baeza-Yates, R., and Marroquín, J. L. Searching in metric spaces. *ACM computing surveys (CSUR) 33*, 3 (2001), 273–321.

[21] Ciaccia, P., Patella, M., and Zezula, P. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB conference, Athens, Greece* (1997), Citeseer, pp. 426–435.

[22] Cole, R. Slowing down sorting networks to obtain faster sorting algorithm. In *25th Annual FOCS Symposium* (1984), IEEE, pp. 255–260.

[23] Cross, P., Heisey, D., Bowers, J., Hay, C., Wolhuter, J., Buss, P., Hofmeyr, M., Michel, A., Bengis, R. G., Bird, T., et al. Disease, predation and demography: assessing the impacts of bovine tuberculosis on african buffalo by monitoring at individual and population levels. *Journal of Applied Ecology 46*, 2 (2009), 467–475.

[24] Dasgupta, S. Lecture notes on k-center clustering., 2013.

[25] De Berg, M., Cook IV, A. F., and Gudmundsson, J. Fast Fréchet queries. *Computational Geometry 46*, 6 (2013), 747–755.

[26] de Berg, M., Gudmundsson, J., and Mehrabi, A. D. A dynamic data structure for approximate proximity queries in trajectory data. In *Proc. of the 25th ACM SIGSPATIAL Conf.* (2017), ACM, p. 48.

[27] Douglas, D. H., and Peucker, T. K. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization 10*, 2 (1973), 112–122.

[28] Driemel, A., and Silvestri, F. Locality-sensitive hashing of curves. In *33rd International Symposium on Computational Geometry, SoCG 2017, July 4-7, 2017, Brisbane, Australia* (2017), pp. 37:1–37:16.

[29] Dütsch, F., and Vahrenhold, J. A filter-and-refinement-algorithm for range queries based on the Fréchet distance (GIS Cup). In *Proceedings of the 25th ACM SIGSPATIAL Conference* (2017), ACM, p. 100.

[30] Eiter, T., and Mannila, H. Computing discrete Fréchet distance. Tech. rep., Citeseer, 1994.

[31] Eppstein, D. Blum-style analysis of quickselect, Oct. 2007.

[32] Frentzos, E., Gratsias, K., Pelekis, N., and Theodoridis, Y. Nearest neighbor search on moving object trajectories. In *SSTD* (2005), Springer, pp. 328–345.

[33] Frey, B. J., and Dueck, D. Clustering by passing messages between data points. *science 315*, 5814 (2007), 972–976.

[34] Gagliardo, A., Pollonara, E., and Wikelski, M. Pigeon navigation: exposure to environmental odours prior release is sufficient for homeward orientation, but not for homing. *Journal of Experimental Biology* (2016), jeb–140889.

[35] Giuggioli, L., McKetterick, T. J., and Holderied, M. Delayed response and biosonar perception explain movement coordination in trawling bats. *PLoS computational biology 11*, 3 (2015), e1004089.

[36] Gonzalez, T. F. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science 38* (1985), 293–306.

[37] Gudmundsson, J., and Smid, M. Fast algorithms for approximate Fréchet matching queries in geometric trees. *Computational Geometry 48*, 6 (2015), 479–494.

[38] Gupta, A., Krauthgamer, R., and Lee, J. R. Bounded geometries, fractals, and low-distortion embeddings. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings* (2003), pp. 534–543.

[39] Guttman, A. *R-trees: A dynamic index structure for spatial searching*, vol. 14. ACM, 1984.

[40] Hetland, M. *The Basic Principles of Metric Indexing. In: Coello C.A.C., Dehuri S., Ghosh S. (eds) Swarm Intelligence for Multi-objective Problems in Data Mining*, vol. 242. Springer, 2009.

[41] Indyk, P. Approximate nearest neighbor algorithms for Fréchet distance via product metrics. In *Proceedings of the eighteenth annual symposium on Computational geometry* (2002), ACM, pp. 102–106.

[42] Kailing, K., Kriegel, H.-P., and Kröger, P. Density-connected subspace clustering for high-dimensional data. In *Proceedings of the 2004 SIAM international conference on data mining* (2004), SIAM, pp. 246–256.

[43] Kalantari, I., and McDonald, G. A data structure and an algorithm for the nearest point problem. *IEEE TSE*, 5 (1983), 631–634.

[44] Karger, D. R., and Ruhl, M. Finding nearest neighbors in growth-restricted metrics. In *Proceedings on 34th Annual ACM STOC, May 19-21, 2002, Montréal, Québec, Canada* (2002), pp. 741–750.

[45] Kays, R., Flowers, J., and Kennedy-Stoskopf, S. Cat tracker project. http://www.movebank.org/, 2016.

[46] Keogh, E., and Ratanamahatana, C. A. Exact indexing of dynamic time warping. *Knowledge and information systems 7*, 3 (2005), 358–386.

[47] Kibriya, A. M., and Frank, E. An empirical comparison of exact nearest neighbour algorithms. In *Knowledge Discovery in Databases: In Proc. of ECML-PKDD* (2007), pp. 140–151.

[48] Kraskov, A., Stögbauer, H., Andrzejak, R. G., and Grassberger, P. Hierarchical clustering using mutual information. *EPL (Europhysics Letters) 70*, 2 (2005), 278.

[49] Krauthgamer, R., and Lee, J. R. Navigating nets: simple algorithms for proximity search. In *Proceedings of the 15th Annual ACM-SIAM SODA* (2004), pp. 798–807.

[50] Li, H., Liu, J., Liu, R. W., Xiong, N., Wu, K., and Kim, T.-h. A dimensionality reduction-based multi-step clustering method for robust vessel trajectory analysis. *Sensors 17*, 8 (2017), 1792.

[51] Long, C., Wong, R. C.-W., and Jagadish, H. Trajectory simplification: On minimizing the direction-based error. *Proceedings of the VLDB Endowment 8*, 1 (2014), 49–60.

[52] Microsoft. Microsoft research asia, GeoLife GPS trajectories. http://www.microsoft.com/en-us/download/details.aspx?id=52367, 2012.

[53] Moret, B. M. Towards a discipline of experimental algorithmics. *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges 59* (2002), 197–213.

[54] NOAA. National hurricane center, national oceanic and atmospheric administration, HURDAT2 atlantic hurricane database. http://www.nhc.noaa.gov/data/, 2017.

[55] Poli, C. L., Harrison, A.-L., Vallarino, A., Gerard, P. D., and Jodice, P. G. Dynamic oceanography determines fine scale foraging behavior of masked boobies in the gulf of mexico. *PloS one 12*, 6 (2017), e0178318.

[56] Shah, R., and Romijnders, R. Applying deep learning to basketball trajectories. *arXiv preprint arXiv:1608.03793* (2016).

[57] STATS. STATS LLC - data science. http://www.stats.com/data-science/, 2015.

[58] Uhlmann, J. K. Metric trees. *Applied Mathematics Letters 4*, 5 (1991), 61–62.

[59] Uhlmann, J. K. Satisfying general proximity/similarity queries with metric trees. *Information processing letters 40*, 4 (1991), 175–179.

[60] Vaserstein, L. N. Markov processes over denumerable products of spaces, describing large systems of automata. *Problemy Peredachi Informatsii 5*, 3 (1969), 64–72.

[61] Vlachos, M., Kollios, G., and Gunopulos, D. Discovering similar multidimensional trajectories. In *Proceedings 18th international conference on data engineering* (2002), IEEE, pp. 673–684.

[62] Weber, R., Schek, H., and Blott, S. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of 24th VLDB Conference* (1998), pp. 194–205.

[63] Wikelski, M., Arriero, E., Gagliardo, A., Holland, R. A., Huttunen, M. J., Juvaste, R., Mueller, I., Tertitski, G., Thorup, K., Wild, M., et al. True navigation in migrating gulls requires intact olfactory nerves. *Scientific reports 5* (2015), 17061.

[64] Williams, B. H., Toussaint, M., and Storkey, A. J. Extracting motion primitives from natural handwriting data. In *ICANN* (2006), Springer, pp. 634–643.

[65] Xie, D., Li, F., and Phillips, J. M. Distributed trajectory similarity search. *Proceedings of the VLDB Endowment 10*, 11 (2017), 1478–1489.

[66] Yianilos, P. N. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the 4th Annual ACM/SIGACT-SIAM SODA Symposium* (1993), pp. 311–321.

[67] Yuan, J., Zheng, Y., Xie, X., and Sun, G. Driving with knowledge from the physical world. In *Proc. of the 17th ACM SIGKDD Conf.* (2011), ACM, pp. 316–324.

[68] Yuan, J., Zheng, Y., Zhang, C., Xie, W., Xie, X., Sun, G., and Huang, Y. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th ACM SIGSPATIAL Conference* (2010), ACM, pp. 99–108.

[69] Zhang, D., Ding, M., Yang, D., Liu, Y., Fan, J., and Shen, H. T. Trajectory simplification: an experimental study and quality analysis. *Proceedings of the VLDB Endowment 11*, 9 (2018), 934–946.

A Construction and Query Runtime

The main focus of this work was to measure the number of distance computations and query I/O, per [40] which underscores that reducing these two measures (especially the first) should dominate algorithm design and experimentation analysis. However, it can also be useful to measure algorithm construction and query runtimes so that one can get a 'ballpark' estimate of how much time is spent. It can also be interesting to see which characteristics impact runtimes and what the trends are.

To this end, Figure 19 shows Relaxed CCT construction and exact query runtimes using synthetic data sets. An increase in cluster size, n, $|\mathcal{S}|$, and d result in increased runtimes. This is expected since increases in these characteristics can result in more δ_F calls and node visits, and increases in n can lead to longer runtimes when computing δ_F and linear bounds.

It is noteworthy that for a given algorithm time complexity, experiment runtimes can vary depending on the underlying hardware and use of software engineering techniques. Indeed, factor speedups can be achieved using approaches such as reducing memory consumption and access, parallelization, caching, using inline functions, multi-threading, or avoiding square root operations. Furthermore, in our setting runtimes are dependent on the choice of distance measure and its implementation details. For example, in this study we used a cubic complexity algorithm that computes δ_F exactly (other approaches such as a divide and conquer search can improve the δ_F time complexity at the expense of precision). For this work, runtimes were not part of core results and so we did not spend effort to improve this measure.

Our experiments were performed on a desktop computer with a 3.60GHz Intel Core i7-7700 CPU, 32GB RAM, running on a Matlab R2018b implementation over a Windows 10 64-bit OS. If better runtimes are a paramount consideration, then a C++ implementation employing similar engineering techniques may significantly improve runtimes.

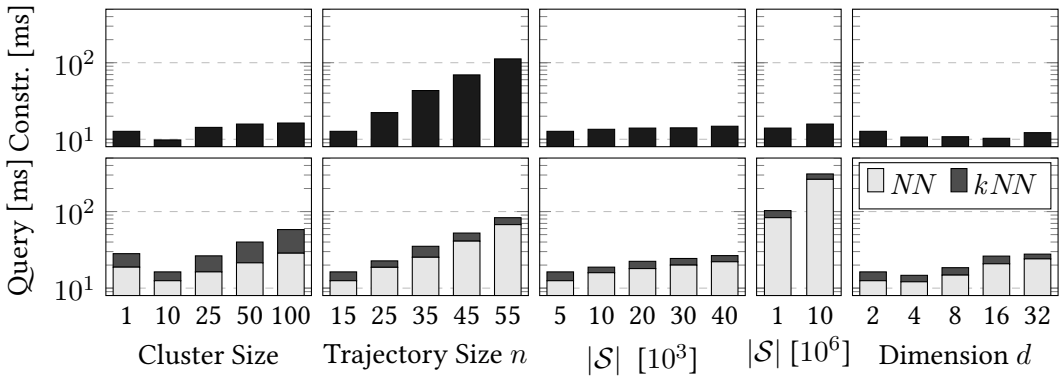

Figure 19: Construction and query runtimes (milliseconds) on synthetic data set Relaxed CCTs. The top shows average construction runtime per trajectory. The bottom shows query latency (end-to-end query runtime) of exact NN (light shade) and kNN $k = 5$ (dark shade) queries, averaged over 1000 queries.