Approximating the Packedness of Polygonal Curves

Joachim Gudmundsson
The University of Sydney, Australia

Yuan Sha
The University of Sydney, Australia

Sampson Wong
The University of Sydney, Australia

Abstract

In 2012 Driemel et al. [17] introduced the concept of c-packed curves as a realistic input model. In the case when c is a constant they gave a near linear time $(1+\varepsilon)$-approximation algorithm for computing the Fréchet distance between two c-packed polygonal curves. Since then a number of papers have used the model.

In this paper we consider the problem of computing the smallest c for which a given polygonal curve in $\mathbb{R}^d$ is c-packed. We present two approximation algorithms. The first algorithm is a 2-approximation algorithm and runs in $O(dn^2 \log n)$ time. In the case $d = 2$ we develop a faster algorithm that returns a $(6+\varepsilon)$-approximation and runs in $O((n/\varepsilon^3)^{4/3} \operatorname{polylog}(n/\varepsilon)))$ time.

We also implemented the first algorithm and computed the approximate packedness-value for 16 sets of real-world trajectories. The experiments indicate that the notion of c-packedness is a useful realistic input model for many curves and trajectories.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Computational geometry, trajectories, realistic input models

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2020.9

Related Version A full version of this paper is available at `https://arxiv.org/abs/2009.07789`.

1 Introduction

Worst-case analysis often fails to accurately estimate the performance of an algorithm for real-world data. One reason for this is that the traditional analysis of algorithms and data structures is only done in terms of the number of elementary objects in the input; it does not take into account their distribution. Problems with traditional analysis have led researchers to analyse algorithms under certain assumptions on the input [15], which are often satisfied in practice. By doing this, complicated hypothetical inputs are hopefully precluded, and the worst-case analysis yields bounds which better reflect the behaviour of the algorithms in practical situations.

In computational geometry, realistic input models were introduced by van der Stappen and Overmars [39] in 1994. They studied motion planning among fat obstacles. Since then a range of models have been proposed, including uncluttered scenes [14], low density [40], simple-cover complexity [33], to name a few. De Berg et al. [15] gave algorithms for computing the model parameters for planar polygonal scenes. In their paper they motivated why such algorithms are important.

- To verify whether a certain model is appropriate for a certain application domain.
- Some algorithms require the value of the model parameter as input in order to work correctly, e.g. the range searching data structure for fat objects developed by Overmars and van der Stappen [35].
- Computing the model parameters of a given input can be useful for selecting the algorithm best tailored to that specific input.

In this paper we will study polygonal curves in $\mathbb{R}^d$. The Fréchet distance [21] is probably the most popular distance measure for curves. In 1995, Alt and Godau [3] presented an $O(n^2 \log n)$ time algorithm for computing the Fréchet distance between two polygonal curves of complexity n. This was later improved by Buchin et al. [29] who showed that the continuous Fréchet distance can be computed in $O(n^2\sqrt{\log n}(\log\log n)^{3/2})$ expected time. Any attempt to find a much faster algorithm was proven to be futile when Bringmann [6] showed that, assuming the Strong Exponential Time Hypothesis, the Fréchet distance cannot be computed in strongly subquadratic time, i.e., in time $O(n^{2-\varepsilon})$ for any $\varepsilon > 0$.

In an attempt to break the quadratic lower bound for realistic curves, Driemel et al. [17] introduced a new family of realistic curves, so-called c-packed curves, which since then has gained considerable attention [10, 16, 18, 27, 28]. A curve π is c-packed if for any ball B, the length of the portion of π contained in B is at most c times the radius of B. In their paper they considered the problem of computing the Fréchet distance between two c-packed curves and presented a $(1+\varepsilon)$-approximation algorithm with running time $O(\frac{cn}{\varepsilon} + cn\log n)$, which was later improved to $O(\frac{cn}{\sqrt{\varepsilon}}\log^2(1/\varepsilon) + cn\log n)$ by Bringmann and Künnemann [7].

Other models for realistic curves have also been studied. Closely related to c-packedness is γ-density which was introduced by Van der Stappen et al. [40] for obstacles, and modified to polygonal curves in [17]. A set of objects is γ-low-density, if for any ball of any radius, the number of objects intersecting the ball that are larger than the ball is less than γ. Aronov et al. [5] studied so-called backbone curves, which are used to model protein backbones in molecular biology. Backbone curves are required to have, roughly, unit edge length and a given minimal distance between any pair of vertices. Alt et al. [4] introduced κ straight curves, which are curves where the arc length between any two points on the curve is at most a constant κ times their Euclidean distance. They also introduced κ-bounded curves which is a generalization of κ-straight curves. It has been shown [2] that one can decide in $O(n\log n)$ time whether a given curve is backbone, κ-straight or κ-bounded.

From the above discussion and the fact that the c-packed model has gained in popularity, we study two natural and important questions in this paper.

1. Given a curve π, how fast can one (approximately) decide the smallest c for which π is c-packed?
2. Are real-world trajectory data c-packed for some reasonable value of c?

Vigneron [41] gave an FPTAS for optimizing the sum of algebraic functions. The algorithm can be applied to compute a $(1+\varepsilon)$ approximation of the c-packedness value of a polygonal curve in $\mathbb{R}^d$ in $O((\frac{n}{\varepsilon})^{d+2}\log^{d+2}\frac{n}{\varepsilon})$ time.

However, working with balls is complicated (see Section 1.1) and in this paper we will therefore consider a simplified version of c-packedness. Instead of balls we will use (d-)cubes, that is, we say that a curve π is c-packed if for any cube S, the length of the portion of π contained in S is at most $c \cdot r$, where r is half the side length of S. Note that under this definition, a c-packed curve using the "ball" definition is a $(\sqrt{d}c)$-packed curve in the "cube"' definition, while a c-packed curve using the "cube" definition is also a c-packed curve in the "ball" definition. From now on we will use the "cube" definition of c-packed curves.

To the best of our knowledge the only known algorithm for computing packedness of a polygonal curve, apart from applying the tool by Vigneron [41], is by Gudmundsson et al. [24] who gave a cubic time algorithm for polygonal curves in $\mathbb{R}^2$. They consider the problem of computing "hotspots" for a given polygonal curve, but their algorithm can also compute the packedness of a polygonal curve. We provide two sub-cubic time approximation algorithms for the packedness of a polygonal curve.

Our first result is a simple $O(dn^2 \log n)$ time 2-approximation algorithm for d-dimensional polygonal curves. We also implemented this algorithm and tested it on 16 data sets to estimate the packedness value for real-world trajectory data. As expected the value varies wildly both between different data sets but also within the same data set. However, about half the data sets had an average packedness value less than 10, which indicates that c-packedness is a useful and realistic model for many real-world data sets.

Our second result is a faster $O^*(n^{4/3})$ time[1] $(6+\varepsilon)$-approximation algorithm for polygonal curves in the plane. We achieve this faster algorithm by applying Callahan and Kosaraju's Well-Separated Pair Decomposition (WSPD) to select $O(n)$ squares, and then approximating the packedness values of these squares with a multi-level data structure. Note that our approach of building a data structure and then performing a linear number of square packedness queries solves a generalised instance of Hopcroft's problem. Hopcroft's problem asks: Given a set of n points and n lines in the plane, does any point lie on a line? An $\Omega(n^{4/3})$ lower bound for Hopcroft's problem was given by Erickson [19]. Hence, it is unlikely that our approach, or a similar approach, can lead to a considerably faster algorithm.

1.1 Preliminaries and our results

Let $\pi = \langle p_1, \ldots, p_n \rangle$ be a polygonal curve in $\mathbb{R}^d$ and let $s_i = (p_i, p_{i+1})$ for $1 \leqslant i < n$. Let H be a closed convex region in $\mathbb{R}^d$. The function $\Upsilon(H) = \sum_{i=1}^{n-1} |s_i \cap H|$ describes the total length of the trajectory π inside H. In the original definition of c-packedness H is a ball.

As mentioned in the introduction, we will consider H to be an axis-aligned cube instead of a ball. The reason for our choice was argued for $\mathbb{R}^2$ in [24], and for completeness we include their arguments here.

If H is a square, then each piece of $\Upsilon(H)$ is a simple linear function, i.e. is of the form $\gamma(x) = ax + b$ for some $a, b \in \mathbb{R}$. The description of each piece of Υ is constant size and can be evaluated in constant time. However, if H is a disc, the intersection points of the boundary of H with the trajectory π are no longer simple linear equations in terms of the center and radius of H, so that Υ becomes a piecewise the sum of square roots of polynomial functions. These square root functions provide algebraic issues that cannot be easily resolved for maximising the function $\Upsilon(H)/r$. For this reason, we will consider H to be a square instead of a disc.

The function $\Upsilon(H) = \sum_{i=1}^{n-1} |s_i \cap H|$ describes the total length of the polygonal curve inside H. Similarly, $\Psi(H) = \Upsilon(H)/r$ denotes the *packedness value* of H. Our aim is to find a cube H^* with centre at p^* and radius r^* that has the maximum packedness value for a given polygonal curve π. The radius of a cube is half the side length of the cube.

The following two theorems summarise the main results of this paper.

▶ **Theorem 1.** *Given a polygonal curve π of size n in $\mathbb{R}^d$, one can compute a 2-approximate packedness value for π in $O(dn^2 \log n)$ time.*

▶ **Theorem 2.** *Given a polygonal curve π of size n in $\mathbb{R}^2$ and a constant ε, with $0 < \varepsilon \leqslant 1$, one can compute a $(6+\varepsilon)$-approximate packedness value for π in $O((n/\varepsilon^3)^{4/3} \operatorname{polylog}(n/\varepsilon))$ time.*

Theorem 1 is presented in Section 2 and Theorem 2 is presented in Section 3. Experimental results on the packedness values for real world data sets are given in Section 2.1.

[1] The O^*-notation omits polylog and $1/\varepsilon$ factors.

2 A 2-approximation algorithm

Given a polygonal curve π in $\mathbb{R}^d$, let H^* be a d-cube with centre at p^* and radius r^* that has a maximum packedness value. Our approximation algorithm builds on two observation. The first observation is that given a center $p \in \mathbb{R}^d$ one can in $O(dn \log n)$ time find, of all possible d-cubes centered at p, the d-cube that has the largest packedness value. The second observation is that there exists a d-cube centered at a vertex of π that has a packedness value that is at least half the packedness value of H^*.

Before we present the algorithm we need some notations. Let H_r^p be the d-cube H, scaled with p as center and such that its radius is r. Fix a point p in $\mathbb{R}^d$, and consider Ψ as a function of r. More formally, let $\psi_p(r) = \Psi(H_r^p)$. Gudmundsson et al. [24] showed properties of $\psi_p(r)$ that we generalize to $\mathbb{R}^d$ and restate as:

▶ **Lemma 3.** *The function $\psi_p(r)$ is a piecewise hyperbolic function. The pieces of $\psi_p(r)$ are of the form $a(1/r) + b$, for $a, b \in \mathbb{R}$, and the break points of $\psi_p(r)$ correspond to d-cubes H where: (i) a vertex of π lies on a $(d-1)$-face of H, or (ii) a $(d-2)$-face (in $\mathbb{R}^3$, an edge) of H intersects an edge of π.*

As a corollary we get:

▶ **Corollary 4.** *Let r_1, r_2 be the radii of two consecutive break points of $\psi_p(r)$, where $r_2 > r_1$. It holds that $\max_{r\in[r_1,r_2]} \psi_p(r) = \max\{\psi_p(r_1), \psi_p(r_2)\}$, that is, the maximum value is obtained either at r_1 or at r_2.*

Proof. According to Lemma 3 the function $\psi_p(r)$ is a hyperbolic function in the range $r \in [r_1, r_2]$ of the form $a(1/r) + b$ with the derivative $-a/r^2$. This implies that $\psi_p(r)$ is a monotonically decreasing or monotonically increasing function in $[r_1, r_2]$. As a result the maximum value of $\psi_p(r)$ is attained either at r_1 or at r_2. ◀

Next we state the algorithm for the first observation. The general idea is to use plane-sweep, scaling d-cube H with centre at p by increasing its radius from 0 to ∞. The $(d-1)$-faces of H are bounded by $2d$ hyperplanes in $\mathbb{R}^d$. When H expands from p, it can first meet a segment of π in one of two ways: (i) a vertex of the segment lies on one of H's $(d-1)$-face, or (ii) an interior point of the segment lies on a $(d-2)$-face of H. For the first case, the new segment can change to intersect a different $(d-1)$-face of H at most $d-1$ times, depending on its relative position to the center of H and its components in all d dimensions.

Similarly for a segment of the second case, it can change to intersect a different $(d-1)$-face of H $O(d)$ times. Thus each segment has $O(d)$ event points and there are $O(dn)$ events in total. Sort the events by their radii $r_1, \ldots, r_m (m = O(dn))$ in increasing order. Perform the sweep by increasing the radius r starting at $r = 0$ and continue until all events have been encountered.

Recall that $\Upsilon(H)$ is the total length of the trajectory π inside H. For each r_i, $1 \leqslant i \leqslant m$, we can compute $\psi_p = \Upsilon(r_i)/r$ in time $O(dn)$. For two consecutive radii r_i and r_{i+1}, $\Upsilon(H_{r_i}^p)$ and $\Upsilon(H_{r_{i+1}}^p)$ can differ in one of three ways. First, $H_{r_{i+1}}^p$ may include a vertex not in $H_{r_i}^p$, in which case the set of contributing edges may increase by up to two. Second, $H_{r_{i+1}}^p$ may intersect an edge not in $H_{r_i}^p$. Finally, an edge in $H_{r_i}^p$ may intersect a different $(d-1)$-face. We can compute a function $\Delta(r_i, r_{i+1})$ that describes these changes in constant time. We then have $\Upsilon(H_{r_{i+1}}^p) = \Upsilon(H_{r_i}^p) + \Delta(r_i, r_{i+1})$, and we can compute $\Upsilon(H_{r_{i+1}}^p)$ from $\Upsilon(H_{r_i}^p)$ in constant time (in $\mathbb{R}^2$ similar to [12]). Apart from sorting the event points, we compute $\psi_p(r)$ for every r_i, $1 \leqslant i \leqslant m$, in $O(dn)$ time. We return radius $\arg\max_{r_1 \leqslant r_i \leqslant r_m} \psi_p(r_i)$ as the result. Hence, the total running time is $O(dn \log n)$.

Note that the break points of $\psi_p(r)$ are the event points. The correctness follows immediately from Corollary 4 which tells us that we only need to consider the set of event points. To summarise we get:

▶ **Lemma 5.** *Given a point p in $\mathbb{R}^d$ one can in $O(dn \log n)$ time determine the radius $r > 0$ such that $\Psi(H_r^p) = \max_{r'>0} \psi_p(r')$.*

Now we are ready to prove the second observation.

▶ **Lemma 6.** *Consider the function $\psi_p(r)$ for a single segment s, i.e. $\psi_p(r) = \frac{|s \cap H_r^p|}{r}$. If the first point on s encountered by H is an interior point of s then the function is non-decreasing from $r = 0$ until H_r^p encounters a vertex of s.*

Proof. The function is zero until an interior point on s is encountered. After encountering the interior point and before encountering a vertex of s, the segment $|s \cap H_r^p|$ is a chord between two boundary points of H_r^p. Suppose we normalise the size of the d-cube H_r^p to be unit-sized. Then the length of the chord is normalised to $\frac{|s \cap H_r^p|}{r} = \psi_p(r)$. Before normalisation, the segment $|s \cap H_r^p|$ had fixed gradient, and had fixed orthogonal distance to the center p. After normalisation, the chord has fixed gradient and has decreasing distance to the center p. Therefore its length $\psi_p(r)$ is non-decreasing as it approaches the diameter of H_r^p. ◀

▶ **Lemma 7.** *There exists a d-cube H with center at a vertex of π such that $\Psi(H) \geq \frac{1}{2} \cdot \Psi(H^*)$, where H^* is the d-cube having the highest packedness value for π.*

Proof. Consider H^*. We will construct a d-cube H that is centered at a vertex p of π and contains H^*. We will then prove that H has packedness value at least $\frac{1}{2} \cdot \Psi(H^*)$, which would prove the theorem. To construct H, we consider two cases:

Case 1: The square H^* does not contain a vertex of π, see Fig. 1(a). Scale H^* until its boundary hits a vertex. Let H_1 denote the d-cube obtained from the scaling and let v be the vertex on the $(d-1)$-face of H_1. According to Lemma 6, we know that $\Psi(H_1) \geq \Psi(H^*)$.

Let H_2 be the d-cube centered at v with radius twice the radius of H_1, as illustrated in Fig. 1(a). Clearly H_2 contains $H_1 \cap \pi$, so $\Psi(H_2) \geq \frac{1}{2}\Psi(H_1) \geq \frac{1}{2}\Psi(H^*)$, as required.

Case 2: The d-cube H^* contains one or more vertices, see Fig. 1(b). Let v be a vertex inside H^*. Let H_2 be the d-cube with center at v and radius twice that of H^*. Again, we have H_2 completely contains $H^* \cap \pi$, so $\Psi(H_2) \geq \frac{1}{2}\Psi(H^*)$, as required.

In both cases, we have constructed a d-cube H_2 centered at a vertex of π for which $\Psi(H_2) \geq \frac{1}{2}\Psi(H^*)$, which proves the lemma. ◀

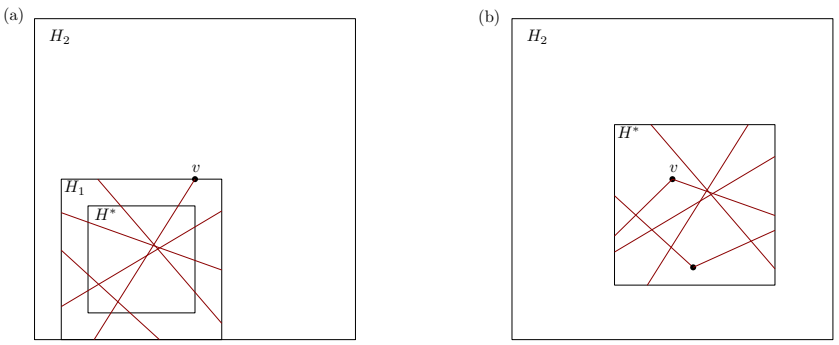

Figure 1 Illustrating the two cases in the proof of Lemma 7: Case 1 in (a) and Case 2 in (b).

Table 1 The table lists 16 real-world data sets. The second and third columns shows the number of curves and the maximum complexity of a curve in the set. The following three columns lists the minimum, maximum and average approximate packedness values. The rightmost column states the average ratio between c and n for the data sets.

Dataset	#Curves	MaxCurveSize	Min	Max	Avg	Avg c/n
Vessel-Y	187	320	2.37	14.28	3.03	0.022
Hurdat	1785	133	2	16.58	3.24	0.154
Pen	2858	182	4.07	20.82	8.79	0.073
Bats	545	736	1.08	29.52	3.60	0.0625
Bus	148	1012	3.21	34.99	14.70	0.052
Vessel-M	103	143	1.04	46.19	4.66	0.272
Basketball	20780	138	2.00	48.65	3.95	0.092
Football	18028	853	2.12	48.87	7.66	0.045
Truck	276	983	5.32	110.44	25.48	0.079
Buffalo	163	479	1.17	254.14	68.42	0.505
Pigeon	131	1504	3.64	275.18	90.93	0.12
Geolife	1000	64390	1.02	858.19	23.31	0.057
Gull	241	3237	1.02	1082.20	139.50	0.478
Cats	152	2257	6.04	1122.77	207.86	0.655
Seabirds	63	2970	5.59	1803.72	825.57	0.388
Taxi	1000	115732	2.20	4255.23	55.38	0.313

2.1 Experimental results

We implemented the above algorithm to test the approximate packedness of real-world trajectory data. We ran the algorithm on 16 data sets. The data sets were kindly provided to us by the authors of [25]. Table 2 summarises the data sets and is taken from [25]. The minimum/maximum/average (approximate) packedness values and the ratio between c and n for each dataset are listed in Table 1. Both the Geolife dataset and the Taxi dataset consist of over 20k trajectories, many of which are very large. For the experiments we randomly sampled 1,000 trajectories from each of these sets.

Although these are only sixteen data sets, it is clear that the notion of c-packedness is a reasonable model for many real-world data sets. For example, the maximal packedness value for all trajectories in all the first eight data sets is less than 50 and the average (approximate) packedness value is below 15. Looking at the ratio between c and n, we can see that for many data sets the value of c is considerably smaller than n.

Consider the task of computing the continuous Fréchet distance between two trajectories. For two trajectories of complexity n, computing the distance will require $O^*(n^2)$ time (even for an $O(1)$-approximation) while a $(1+\varepsilon)$-approximation can be obtained for c-packed trajectories in $O^*(cn)$ time. Thus the algorithm by Driemel et al. [17] for c-packed curves is likely to be more efficient than the general algorithm for these data sets.

3 A fast $(6+\varepsilon)$-approximation algorithm

In this section we will take a different approach to Section 2 to yield an algorithm that considers a linear number of squares rather than a quadratic number of squares. First we will identify a set $\mathcal{S}$ containing a linear number of squares that will include a square having a high packedness value (Section 3.1). Then we will build a multi-level data structure (Section 3.2) on π such that given a square $S \in \mathcal{S}$ it can quickly approximate $|S \cap \pi|$.

Table 2 Real data sets, showing number of input trajectories n, dimensions d, average number of simplified vertices per trajectory, and a description.

Data Set	n	d	#vertices	Trajectory Description
Vessel-M [31]	106	2	23.0	Mississippi river shipping vessels Shipboard AIS.
Pigeon [22]	131	2	970.0	Homing Pigeons (release sites to home site).
Seabird [36]	134	2	3175.8	GPS of Masked Boobies in Gulf of Mexico.
Bus [20]	148	2	446.6	GPS of School buses.
Cats [30]	154	2	526.1	Pet house cats GPS in Raleigh-Durham, NC, USA.
Buffalo [11]	165	2	161.3	Radio-collared Kruger Buffalo, South Africa.
Vessel-Y [31]	187	2	155.2	Yangtze river shipping Vessels Shipboard AIS.
Gulls [42]	253	2	602.1	Black-backed gulls GPS (Finland to Africa).
Truck [20]	276	2	406.5	GPS of 50 concrete trucks in Athens, Greece.
Bats [23]	545	2	44.1	Video-grammetry of Daubenton trawling bats.
Hurdat2 [34]	1788	2	27.7	Atlantic tropical cyclone and sub-cyclone paths.
Pen [43]	2858	2	119.8	Pen tip characters on a WACOM tablet.
Football [38]	18034	2	203.4	European football player (team ball-possession).
Geolife [32]	18670	2	1332.5	People movement, mostly in Beijing, China.
Basketball [37]	20780	3	44.1	NBA basketball three-point shots-on-net.
Taxi [44, 45]	180736	2	343.0	10,357 Partitioned Beijing taxi trajectories.

3.1 Linear number of good squares

To prove that it suffices to consider a linear number of squares we will use the well-known Well-Separated Pair Decomposition (WSPD) by Callahan and Kosaraju [8].

Let A and B be two finite sets of points in $\mathbb{R}^d$ and let $s > 0$ be a real number. We say that A and B are well-separated with respect to s, if there exist two disjoint balls C_A and C_B, such that (1) C_A and C_B have the same radius, (2) C_A contains the bounding box of A and C_B contains the bounding box of B, and (3) the distance between C_A and C_B is at least s times the radius of C_A and C_B. The real number s is called the separation ratio.

▶ **Lemma 8.** *Let $s > 0$ be a real number, let A and B be two sets in $\mathbb{R}^d$ that are well-separated with respect to s, let a and a' be two points in A, and let b and b' be two points in B. Then (1) $|aa'| \leq (2/s) \cdot |ab|$, and (2) $|a'b'| \leq (1 + 4/s) \cdot |ab|$.*

▶ **Definition 9.** *Let S be a set of n points in $\mathbb{R}^d$, and let $s > 0$ be a real number. A well-separated pair decomposition (WSPD) for S, with respect to s, is a sequence $\{A_1, B_1\}, \ldots, \{A_m, B_m\}$ of pairs of non-empty subsets of S, for some integer m, such that:*

1. *for each i with $1 \leq i \leq m$, A_i and B_i are well-separated with respect to s, and*
2. *for any two distinct points p and q of S, there is exactly one index i with $1 \leq i \leq m$, such that $p \in A_i$ and $q \in B_i$, or $p \in B_i$ and $q \in A_i$. The integer m is called the size of the WSPD.*

▶ **Lemma 10.** *(Callahan and Kosaraju [8]) Given a set V of n points in $\mathbb{R}^d$, and given a real number $s > 0$, a well-separated pair decomposition for V, with separation ratio s, consisting of $O(s^d n)$ pairs, can be computed in $O(n \log n + s^d n)$ time.*

Now we are ready to construct a set $\mathcal{S}$ of squares. Compute a well-separated pair decomposition $W = \{(A_1, B_1), \ldots, (A_m, B_m)\}$ with separation constant $s = 720/\varepsilon$ for the vertex set of π. For every well-separated pair $(A_i, B_i) \in W$, $1 \leq i \leq k$, construct two squares that will be added to $\mathcal{S}$ as follows:

Pick an arbitrary point $a \in A_i$ and an arbitrary point $b \in B_i$. Construct one square with center at a and radius r, and one square with center at b and radius r, where $r = \max\{|a.x - b.x|, |a.y - b.y|\} + \varepsilon/120 \cdot |ab|$. The two squares are added to $\mathcal{S}$.

It follows immediately from Lemma 10 that the number of squares in $\mathcal{S}$ is $O(n/\varepsilon^2)$ and that one can construct $\mathcal{S}$ in $O(n \log n + n/\varepsilon^2)$ time.

To prove the approximation factor of the algorithm we will first need the following technical lemma. The proof can be found in the full version of the paper [26].

▶ **Lemma 11.** *Let $H^p_{r_1}$ and $H^p_{r_2}$, with $r_2 > r_1$, be two squares with centre at p such that $H^p_{r_2} \setminus H^p_{r_1}$ contains no vertices of π in its interior. For any value r_x, with $r_1 \leqslant r_x \leqslant r_2$, it holds that $\psi_p(r_x) \leq \psi_p(r_1) + 2 \cdot \psi_p(r_2)$.*

Due to Lemma 7, it suffices to consider squares with center at a vertex of π to obtain a 2-approximation. Combining this with Lemma 11, it suffices to consider squares with center at a vertex of π and a vertex of π on its boundary to obtain a 6-approximation. Using the WSPD argument we have reduced our set of squares to a linear number of squares and we will now argue that $\mathcal{S}$ must contain a square that has a high packedness factor. Let H^* be a square with a maximum packedness value of π.

▶ **Lemma 12.** *There exists a square $S \in \mathcal{S}$ such that $\Psi(S) \geqslant \frac{1}{(6+\varepsilon/8)} \cdot \Psi(H^*)$.*

Proof. From Lemmas 7 and 11 we know that there exists a square H with centre at a vertex p of π and whose boundary contains a vertex q of π such that $\Psi(H) \geq \frac{1}{6} \cdot \Psi(H^*)$. According to the construction of $\mathcal{S}$ there exists a square $S \in \mathcal{S}$ such that S has its centre at a point $a \in A_i$ and has radius $r = \max\{|a.x - b.x|, |a.y - b.y|\} + \varepsilon/120 \cdot |ab|$, where b is a point in B_i.

By Lemma 8, we have $|ap| \leq \varepsilon/360 \cdot |ab|$ and $|bq| \leq \varepsilon/360 \cdot |ab|$. If r_H is the radius of H, then $r_H = \max\{|p.x - q.x|, |p.y - q.y|\} \leq \max\{|a.x - b.x|, |a.y - b.y|\} + |ap| + |bq| \leq \max\{|a.x - b.x|, |a.y - b.y|\} + \varepsilon/120 \cdot |ab| - |ap| = r - |ap|$. But p is at most $|ap|$ away from a in both the x and y directions, so H must be entirely contained inside S. So $\Upsilon(S) \geq \Upsilon(H)$.

Next, we show S is not too much larger than H:

$$r = \max\{|a.x - b.x|, |a.y - b.y|\} + \frac{\varepsilon}{120} \cdot |ab| \leqslant r_H + |ap| + |bq| + \frac{\varepsilon}{120} \cdot |ab| \leqslant r_H + \frac{\varepsilon}{72} \cdot |ab|$$
$$\leqslant r_H + \frac{\varepsilon}{72} \cdot (1 + \frac{\varepsilon}{180}) \cdot |pq| \leqslant r_H + \frac{\varepsilon}{36\sqrt{2}} \cdot (1 + \frac{\varepsilon}{180}) \cdot r_H \leqslant (1 + \frac{\varepsilon}{48}) \cdot r_H.$$

Putting this all together yields:

$$\Psi(S) = \Upsilon(S)/r \geq \frac{1}{(1 + \varepsilon/48)} \cdot \Upsilon(H)/r_H = \frac{1}{1 + \varepsilon/48} \cdot \Psi(H) \geq \frac{1}{6 + \varepsilon/8} \cdot \Psi(H^*),$$

which completes the lemma. ◀

3.2 Data structure

The aim of this section is to develop an efficient data structure on π such that queried with an axis-aligned square $S \in \mathcal{S}$ the data structure returns an approximation of $|S \cap \pi|$.

The general idea of the multi-level data structure is that the first level is a modified 1D segment tree, similar to the hereditary segment tree [9]. We partition the set of π's segments into four sets depending on their slope; $(-\infty, -1)$, $[-1, 0)$, $[0, 1)$ and $[1, \infty)$. In the rest of this section we will describe the data structure for the set of segments with slope in $[0, 1)$. The remaining three sets are handled symmetrically.

3.2.1 Modified 1D segment tree

The description of the segment tree follows the description in [13]. Let L be the set of line segments in π. For the purpose of the 1D segment tree, we can view L as a set of intervals on the line. Let $p_1, \ldots, p_m$ be the list of distinct interval endpoints, sorted from left to right. Consider the partitioning of the real line induced by those points. The regions of this partitioning are called *elementary intervals*. Thus, the elementary intervals are, from left to right: $(-\infty, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \ldots, (p_{m-1}, p_m), [p_m, p_m], (p_m, +\infty)$.

Given a set I of intervals, or segments, a segment tree $\mathcal{T}$ for I is structured as follows:

1. $\mathcal{T}$ is a binary tree.
2. Its leaves correspond to the elementary intervals induced by the endpoints in I. The elementary interval corresponding to a leaf v is denoted $Int(v)$.
3. The internal nodes of $\mathcal{T}$ correspond to intervals that are the union of elementary intervals: the interval $Int(N)$ corresponding to an internal node N is the union of the intervals corresponding to the leaves of the tree rooted at N. That implies that $Int(N)$ is the union of the intervals of its two children.
4. Each node or leaf v in $\mathcal{T}$ stores the interval $Int(v)$ and a set of intervals, in some data structure. This canonical subset of node v contains the intervals $[x, x']$ from I such that $[x, x']$ contains $Int(v)$ and does not contain $Int(parent(v))$. That is, each node in $\mathcal{T}$ stores the set of segments $F(v)$ that span through its interval, but do not span through the interval of its parent.

The 1D segment tree can be built in $O(n \log n)$ time, using $O(n \log n)$ space and point stabbing queries can be answered in $O(\log n + k)$ time, where k is the number of segments intersecting the query point.

We make one minor change to $\mathcal{T}$ that will increase the space usage to $O(n \log^2 n)$ but it will allow us to speed up interval stabbing queries. Each internal node v store, apart from the set $F(v)$, all the segments stored in the subtree rooted at v, including $F(v)$. We denote this set by $L(v)$.

The main benefit of this minor modification is that when an interval stabbing query is performed only $O(\log n)$ canonical subsets are required to identify all the segments intersecting the interval. Next we show how to build associated data structures for $L(v)$ and $F(v)$ for each internal node v in $\mathcal{T}$.

3.2.2 Three associated data structures

Consider querying the segment tree $\mathcal{T}$ with a square $S \in \mathcal{S}$. There are three different cases that can occur, and for each of these cases we will build an associated data structure. That is, each internal node will have three types of associated data structures. Consider a query $S = [x, x'] \times [y, y']$ and let μ_l and μ_r be the leaf nodes in $\mathcal{T}$ where the search for the boundary values x and x' end. See Figure 2 for an illustration of the search and the three cases. An internal node v is one of the following types:

Type A: if $Int(v) \subseteq [x, x']$,

Type B: if $Int(v) \cap [x, x'] \neq \emptyset$, $Int(v) \not\subseteq [x, x']$ and $[x, x'] \not\subseteq Int(v)$, or

Type C: if $[x, x'] \subset Int(v)$.

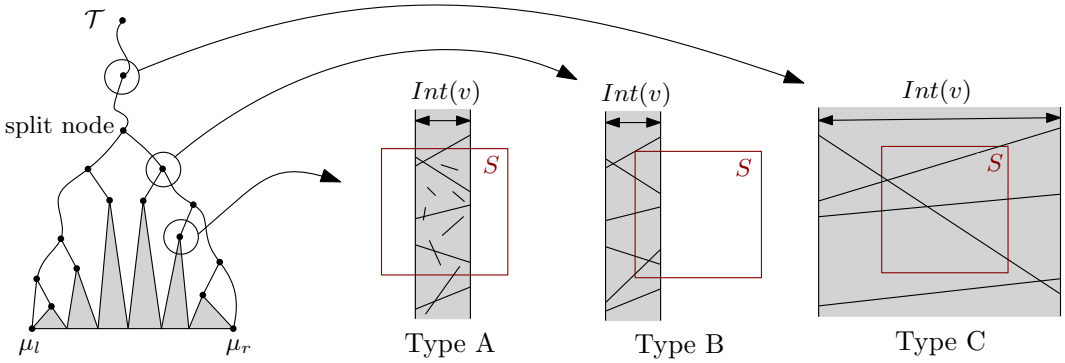

Figure 2 The primary tree $\mathcal{T}$, and the three types of nodes in $\mathcal{T}$ that can be encountered during a query.

Associated data structure for Type A nodes

For a Type A node we need to compute the length of all segments stored in the subtree with root v in the y-interval $[y, y']$. Let $s_1, \ldots, s_m$ be the set of m segments stored in $L(v)$, and let $Y = \langle y_1, \ldots, y_{2m} \rangle$ denote the y-coordinates of the endpoints of the segments in $L(v)$ ordered from bottom-to-top. To simplify the description we assume that the values are distinct.

Let $\delta(y)$ denote the total length of the segments in $L(v)$ below y. For two consecutive y-values y_i and y_{i+1}, the set of edges contributing to $\delta(y_i)$ and $\delta(y_{i+1})$ has increased or decreased by one. So, we can compute a function $\Delta(y_i, y_{i+1})$ that describes these changes in constant time. We then have $\delta(y_{i+1}) = \delta(y_i) + \Delta(y_i, y_{i+1})$, and thus we can compute $\delta(y_{i+1})$ from $\delta(y_i)$ in constant time after sorting the events. Hence, we can compute all the $\delta(y_i)$-values and all the $\Delta(y_i, y_{i+1})$ in time $O(m \log m)$.

Given a y-value y' one can compute $\delta(y')$ as $\delta(y_i) + \frac{y'-y_i}{y_{i+1}-y_i} \cdot \Delta(y_i, y_{i+1})$, where y_i is the largest y-value in Y smaller than y'. Hence, our associated data structure for Type A nodes is a binary tree with respect to the values in Y, where each leaf stores the value y_i, $\delta(y_i)$ and $\Delta(y_i, y_{i+1})$. The tree can be computed in $O(m \log m)$ time using linear space, and can answer queries in $O(\log m)$ time.

▶ **Lemma 13.** *The associated data structures for Type A nodes in $\mathcal{T}$ can be constructed in $O(n \log^2 n)$ time using $O(n \log^2 n)$ space. Given a query square S for an associated data structure of Type A stored in an internal node v, the value $|L(v) \cap S \cap Int(v)|$ is returned in time $O(\log n)$.*

Associated data structure for Type B nodes

The associated data structure for a Type B node is built to handle the case when the query square $S = [x, x'] \times [y, y']$ intersects either the left boundary (x_l) or the right boundary (x_r) of $Int(v)$, but not both. The two cases are symmetric and we will only describe the case when S intersects the right boundary.

The data structure returns a value M that is an upper bound on the length of the segments of $F(v)$ within S and a lower bound on the segments within S^+, where S^+ is a slightly expanded version of S. See Figure 3(c). Formally, $S^+ = [x - \varepsilon/8 \cdot |x_r - x|, x' + \varepsilon/8 \cdot |x_r - x|] \times [y - \varepsilon/8 \cdot |x_r - x|, y' + \varepsilon/8 \cdot |x_r - x|]$.

If S^+ spans $Int(v)$ then we need to use a binary tree on $F(v)$ to answer the query in logarithmic time, similar to the associated data structure for Type A nodes.

If S^+ does not span $Int(v)$ then the query is performed on the Type B associated data structures. We first show how to construct these data structures, then we show how to handle the query. Recall that all the segments in $F(v)$ span the interval $Int(v)$. Let $s_1, \ldots, s_m$ be the set of m segment in $F(v)$ and let $\mu(s_i)$ be the angle of inclination[2] of s_i. The angle of inclination for segments with slope in the interval $[0, 1)$ is in the interval $[0, \pi/4)$. Partition $F(v)$ into κ_1 sets $F_1(v), \ldots, F_{\kappa_1}(v)$ such that for any segment $s_i \in F_j(v)$ it holds that $(j-1) \cdot \frac{\pi}{4\kappa_1} \leq \mu(s_i) < j \cdot \frac{\pi}{4\kappa_1}$.

Consider one such partition $F_j(v) = \{s_1^j, \ldots, s_{m_j}^j\}$. Build a balanced binary search tree T_r^j on the y-coordinates of the right endpoints of the segments in $F_j(v)$. The data structure can be constructed in $O(m_j \log m_j)$ time using linear space. Given a y-interval ℓ as a query, the number of right endpoints in T_r^j within ℓ can be reported in time $O(\log m_j)$. From the above description and the fact that $\sum_{v \in T} F(v) = O(n \log n)$ it immediately follows that the total construction time for all the Type B nodes is $O(n \log^2 n)$ and the total amount of space required is $O(n \log n)$. This completes the construction of the data structures on $F(v)$.

It remains to show how to handle a query, i.e. how to compute M. We focus first on computing an M that upper bounds $|F(v) \cap S|$, and we later prove that M lower bounds $|F(v) \cap S^+|$. There are two steps in computing M. The first step is to count the number of segments that intersect S. The second step is to multiply this count by the *maximum* possible length of intersection between the segment and S. This would clearly yield an upper bound on $|F(v) \cap S|$. To obtain suitable maximum lengths in the second step, we need to subdivide the partitions F_j further.

The right endpoints must lie in a y-interval given by $I_j = [y, y' + \bar{y}]$, where y, y' are the y-coordinates of the bottom and top boundaries of S, and $\bar{y} = (x_r - x) \cdot \tan(\frac{j \cdot \pi}{4\kappa_1})$. Subdivide I_j into three subintervals: $I_j^1 = [y, y + \bar{y})$, $I_j^2 = [y + \bar{y}, y')$ and $I_j^3 = [y', y' + \bar{y})$, see Fig. 3(b). Further subdivide I_j^1 and I_j^3 into $2\kappa_2$ subintervals of $\ell_1^1, \ldots, \ell_{\kappa_2}^1$ and $\ell_1^3, \ldots, \ell_{\kappa_2}^3$ of equal length. Hence we partitioned I_j into a set L_j of $2\kappa_2 + 1$ subintervals.

Given these subdivisions L_j, our first step is to simply perform a range counting query in T_r^j for each $\ell \in L_j$. Our second step is to multiply this count by the maximum length of intersection between S and any segment in F_j with its right endpoint in ℓ. The product of these two values is clearly an upper bound on the length of intersection between S and segments in F_j with right endpoint in ℓ. Finally, we sum over all subdivisions $\ell \in F_j$ and then over all partitions F_j to obtain a value M that upper bounds $|S \cap F(v)|$. The time required to handle a query is $O(\kappa_1 \cdot \kappa_2 \cdot \log m)$. It remains only to prove $M \leq |F(v) \cap S^+|$.

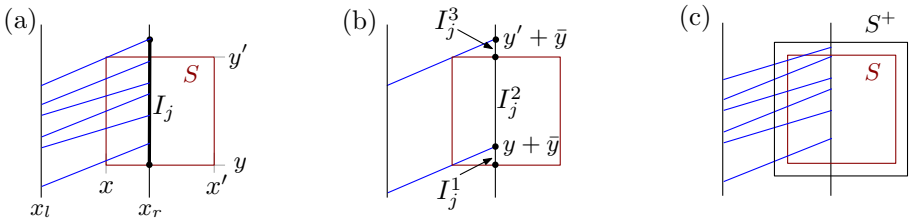

Figure 3 (a) A query S and the set $F(v)$. (b) Illustrating the three interval I_j^1, I_j^2 and I_j^3. (c) The expanded square S^+.

By setting $\kappa_1 = 16\sqrt{2}/\varepsilon$ and $\kappa_2 = 16/\varepsilon$ we can prove the following (for proof, see the full version [26]).

[2] $\mu(s_i)$ is the arctan of the slope in the interval $[0, 1)$.

▶ **Lemma 14.** $M \leq |F(v) \cap S^+|$

We summarise the associated data structures for Type B nodes with the following lemma.

▶ **Lemma 15.** *The associated data structure for Type B nodes can be constructed in $O(n \log^2 n)$ time using $O(n \log n)$ space. Given a query square $S = [x, x'] \times [y, y']$ for an associated data structure of Type B stored in an internal node v, a real value $M(v)$ is returned in $O(\frac{1}{\varepsilon^2} \cdot \log n)$ time such that:*

$$|F(v) \cap S| \leq M(v) \leq |F(v) \cap S^+|,$$

where $S^+ = ([x - \varepsilon/8 \cdot w, x' + \varepsilon/8 \cdot w]) \times [y - \varepsilon/8 \cdot w, y' + \varepsilon/8 \cdot w]$ and w is the width of $S \cap Int(v)$.

Associated data structure for Type C nodes

The associated data structure for Type C nodes have some similarities with the associated data structures for Type B nodes, however, it is a much harder case since we cannot use the ordering on the y-coordinates of the right endpoints like for the Type B nodes. Instead we will precompute an approximation of $|F(v) \cap S|$ for every internal node v in $\mathcal{T}$ and every square $S \in \mathcal{S}$ that lies entirely within $Int(v)$. Recall from Section 3.1 that $\mathcal{S}$ is a set of size $O(n/\varepsilon^2)$ that is guaranteed to have a square that is a $(6 + \varepsilon/8)$-approximation.

Let $\mathcal{S}(v)$ denote the subset of squares in $\mathcal{S}$ that lie entirely within $Int(v)$. The stored value for a square $S \in \mathcal{S}(v)$ is an upper bound on $|F(v) \cap S|$ and a lower bound on $|F(v) \cap S^+|$, where S^+ is the same as the one defined for Type B nodes. See Figure 3(c). If S^+ intersects the left or right boundary of $Int(v)$ then perform the query as a Type B node instead of a Type C node.

The associated data structure will be built on the set $F(v)$, hence, all segments will span the interval $Int(v)$. Let $s_1, \ldots, s_m$ be the segments in $F(v)$ and let $\mu(s_i)$ be the angle of inclination of s_i. Partition $F(v)$ into κ_1 sets $F_1(v), \ldots, F_{\kappa_1}(v)$ such that for any segment $s_i \in F_j(v)$ it holds that $(j-1) \cdot \frac{\pi}{4\kappa_1} \leq \mu(s_i) < j \cdot \frac{\pi}{4\kappa_1}$, with $\kappa_1 = 16\sqrt{2}/\varepsilon$.

Using a combination of the approach we used for Type B nodes and a result by Agarwal [1] we can prove the following lemma (proof available in the full version [26]).

▶ **Lemma 16.** *Given a set $F_j(v) = \{s_1^j, \ldots, s_{n_1}^j\}$ of line segments (as defined above) and a set $\mathcal{S}(v) = \{S_1, \ldots, S_{n_2}\}$ of squares lying entirely within $Int(v)$, one can compute, in $O((n_1 + n_2/\varepsilon)^{4/3} \operatorname{polylog}(n_1 + n_2/\varepsilon))$ time using $O((n_1 + n_2/\varepsilon)^{4/3} / \log^{(2w+1)/3}(n_1 + n_2/\varepsilon))$ space, where w is a constant < 3.33, a set of n_2 real values $\{M_1^j(v), \ldots, M_{n_2}^j(v)\}$ such that for every i, $1 \leq i \leq n_2$ the following holds:*

$$|F_j(v) \cap S_i| \leq M_i^j(v) \leq |F_j(v) \cap S_i^+|.$$

For each set $F_j(v)$ apply Lemma 16, and for each square $S_i \in S(v)$ precompute the value $M_i = \sum_{j=1}^{\kappa_1} M_i^j(v)$. Store all the squares lying within the interval $Int(v)$ in a balanced binary search tree along with their precomputed values M_i. Note that each square of $\mathcal{S}$ can appear at most once on each level of the primary segment tree structure $\mathcal{T}$. Furthermore, an edge $s \in \pi$ can straddle at most two intervals on a level, as a result we get that the total amount of time spent on one level of the segment tree to build the Type C associated data structures is $O((n/\varepsilon^3)^{4/3} \log^{(w+2)/3}(n/\varepsilon))$. Since the number of levels in $\mathcal{T}$ is $O(\log n)$ the total time required to build all the Type C associated data structures is $O((n/\varepsilon^3)^{4/3} \log^{(w+5)/3}(n/\varepsilon))$. Putting all the pieces together we get:

▶ **Lemma 17.** *The Type C associated data structures for $\mathcal{T}$ can be computed in time $O((n/\varepsilon^3)^{4/3}\,\mathrm{polylog}(n/\varepsilon))$ using $O(n^{4/3}/\varepsilon^4)$ space. Given a query square $S \in \mathcal{S}(v)$, a value $M(v)$ can be returned in $O(\log(n/\varepsilon^2))$ time such that:*

$$|F(v) \cap S| \leq M(v) \leq |F(v) \cap S^+|.$$

3.2.3 Putting it together

In the previous section we showed how to construct a two-level data structure that uses a modified segment tree as the primary tree, and a set of associated data structures for all the internal nodes in the primary tree. The primary tree requires $O(n \log n)$ space, and the complexity of the associated data structures is dominated by the Type C nodes, that require $O((n/\varepsilon^3)^{4/3} \log^{(w+5)/3}(n/\varepsilon))$ time and $O((n/\varepsilon^3)^{4/3}/\log^{(2w+1)/3}(n/\varepsilon))$ space to construct. Given a query square $S \in \mathcal{S}$ a value M is returned in $O(\frac{1}{\varepsilon^2} \cdot \log^2 n)$ time such that $\Upsilon(S) \leq M \leq \Upsilon(S^+)$.

According to Lemma 12 there exists a square $S \in \mathcal{S}$ that has a packedness value that is within a factor of $(6 + \varepsilon/8)$ smaller than the maximum packedness value of π. Using the data structure described in this section we get a $(6 + \varepsilon/8)(1 + \varepsilon/8)$-approximation. Since ε is assumed to be at most 1, we finally get Theorem 2.

4 Concluding remarks

In this paper we gave two approximation algorithm for the packedness value of a polygonal curve. The obvious question is if one can get a fast and practical $(1 + \varepsilon)$-approximation.

We also computed approximate packedness values for 16 real-world data sets, and the experiments indicate that the notion of c-packedness is a useful realistic input model for curves and trajectories.

References

1. Pankaj K. Agarwal. Partitioning arrangements of lines II: applications. *Discrete & Computational Geometry*, 5:533–573, 1990. doi:10.1007/BF02187809.
2. Pankaj K. Agarwal, Rolf Klein, Christian Knauer, Stefan Langerman, Pat Morin, Micha Sharir, and Michael Soss. Computing the detour and spanning ratio of paths, trees, and cycles in 2D and 3D. *Discrete & Computational Geometry*, 39(1):17–37, 2008.
3. H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *International Journal of Computational Geometry*, 5:75–91, 1995.
4. Helmut Alt, Christian Knauer, and Carola Wenk. Comparison of distance measures for planar curves. *Algorithmica*, 38(1):45–58, 2004.
5. Boris Aronov, Sariel Har-Peled, Christian Knauer, Yusu Wang, and Carola Wenk. Fréchet distance for curves, revisited. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 52–63, 2006.
6. Karl Bringmann. Why walking the dog takes time: Frechet distance has no strongly subquadratic algorithms unless SETH fails. In *55th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 661–670, 2014.
7. Karl Bringmann and Marvin Künnemann. Improved approximation for Fréchet distance on c-packed curves matching conditional lower bounds. *International Journal on Computational Geometry and Applications*, 27(1-2):85–120, 2017.
8. P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90, 1995.

9 Bernard Chazelle. Reporting and counting segment intersections. *Journal of Computer and System Sciences*, 32(2):156–182, 1986.

10 Daniel Chen, Anne Driemel, Leonidas J. Guibas, Andy Nguyen, and Carola Wenk. Approximate map matching with respect to the Fréchet distance. In *Proceedings of the 13th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 75–83, 2011.

11 P. C. Cross, D. M. Heisey, J. A. Bowers, C. T. Hay, J. Wolhuter, P. Buss, M. Hofmeyr, A. L. Michel, R. G. Bengis, T. L. F. Bird, , 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):467–475, 2009.

12 R. Silverman D. Mount and A. Wu. On the area of overlap of translated polygons. *Computer Vision and Image Understanding*, 64(1):53–61, 1996.

13 M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008.

14 Mark de Berg. Linear size binary space partitions for uncluttered scenes. *Algorithmica*, 28(3):353–366, 2000.

15 Mark de Berg, Frank van der Stappen, Jules Vleugels, and Matya Katz. Realistic input models for geometric algorithms. *Algorithmica*, 34(1):81–97, 2002.

16 Anne Driemel and Sariel Har-Peled. Jaywalking your dog: Computing the Fréchet distance with shortcuts. *SIAM Journal on Computing*, 42(5):1830–1866, 2013.

17 Anne Driemel, Sariel Har-Peled, and Carola Wenk. Approximating the Fréchet distance for realistic curves in near linear time. *Discrete & Computational Geometry*, 48(1):94–127, 2012.

18 Anne Driemel and Amer Krivosija. Probabilistic embeddings of the Fréchet distance. In *Proceedings of the 16th International Workshop Approximation and Online Algorithms*, pages 218–237, 2018.

19 Jeff Erickson. On the relative complexities of some geometric problems. In *Proceedings of the 7th Canadian Conference on Computational Geometry*, pages 85–90. Carleton University, Ottawa, Canada, 1995. URL: `http://www.cccg.ca/proceedings/1995/cccg1995_0014.pdf`.

20 Elias Frentzos, Kostas Gratsias, Nikos Pelekis, and Yannis Theodoridis. Nearest neighbor search on moving object trajectories. In *SSTD*, pages 328–345. Springer, 2005.

21 M. Maurice Fréchet. Sur quelques points du calcul fonctionnel. *Rendiconti del Circolo Matematico di Palermo*, 22:1–72, 1906. `doi:10.1007/BF03018603`.

22 Anna Gagliardo, Enrica Pollonara, and Martin Wikelski. Pigeon navigation: exposure to environmental odours prior release is sufficient for homeward orientation, but not for homing. *Journal of Experimental Biology*, pages jeb–140889, 2016.

23 Luca Giuggioli, Thomas J McKetterick, and Marc Holderied. Delayed response and biosonar perception explain movement coordination in trawling bats. *PLoS computational biology*, 11(3):e1004089, 2015.

24 J. Gudmundsson, M. J. van Kreveld, and F. Staals. Algorithms for hotspot computation on trajectory data. In *21st SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 134–143. ACM, 2013.

25 Joachim Gudmundsson, Michael Horton, John Pfeifer, and Martin Seybold. A practical index structure supporting Fréchet proximity queries among trajectories. arXiv:2005.13773.

26 Joachim Gudmundsson, Yuan Sha, and Sampson Wong. Approximating the packedness of polygonal curves, 2020. `arXiv:2009.07789`.

27 Joachim Gudmundsson and Michiel H. M. Smid. Fast algorithms for approximate Fréchet matching queries in geometric trees. *Computational Geometry*, 48(6):479–494, 2015.

28 Sariel Har-Peled and Benjamin Raichel. The Fréchet distance revisited and extended. *ACM Transactions on Algorithms*, 10(1), 2014.

29 W. Meulemans K. Buchin, M. Buchin and W. Mulzer. Four soviets walk the dog – with an application to alt's conjecture. *Discrete & Computational Geometry*, 58(1):180–216, 2017.

30 Roland Kays, James Flowers, and Suzanne Kennedy-Stoskopf. Cat tracker project. `http://www.movebank.org/`, 2016.

31 Huanhuan Li, Jingxian Liu, Ryan Wen Liu, Naixue Xiong, Kefeng Wu, and Tai-hoon Kim. A dimensionality reduction-based multi-step clustering method for robust vessel trajectory analysis. *Sensors*, 17(8):1792, 2017.

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

33 Joseph S. B. Mitchell, David M. Mount, and Subhash Suri. Query-sensitive ray shooting. *International Journal on Computational Geometry and Applications*, 7(4):317–347, 1997.

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

35 Mark H. Overmars and A. Frank van der Stappen. Range searching and point location among fat objects. *Journal of Algorithms*, 21(3):629–656, 1996.

36 Caroline L Poli, Autumn-Lynn Harrison, Adriana Vallarino, Patrick D Gerard, and Patrick GR Jodice. Dynamic oceanography determines fine scale foraging behavior of masked boobies in the gulf of mexico. *PloS one*, 12(6):e0178318, 2017.

37 Rajiv Shah and Rob Romijnders. Applying deep learning to basketball trajectories. *arXiv preprint arXiv:1608.03793*, 2016.

38 STATS. STATS LLC - data science. `http://www.stats.com/data-science/`, 2015.

39 Frank van der Stappen and Mark H. Overmars. Motion planning amidst fat obstacles (extended abstract). In *Proceedings of the 10th Annual Symposium on Computational Geometry*, pages 31–40, 1994.

40 Frank van der Stappen, Mark H. Overmars, Mark de Berg, and Jules Vleugels. Motion planning in environments with low obstacle density. *Discrete & Computational Geometry*, 20(4):561–587, 1998.

41 Antoine Vigneron. Geometric optimization and sums of algebraic functions. *ACM Trans. Algorithms*, 10(1):4:1–4:20, 2014. `doi:10.1145/2532647`.

42 Martin Wikelski, Elena Arriero, Anna Gagliardo, Richard A Holland, Markku J Huttunen, Risto Juvaste, Inge Mueller, Grigori Tertitski, Kasper Thorup, Martin Wild, et al. True navigation in migrating gulls requires intact olfactory nerves. *Scientific reports*, 5:17061, 2015.

43 Ben H Williams, Marc Toussaint, and Amos J Storkey. Extracting motion primitives from natural handwriting data. In *ICANN*, pages 634–643. Springer, 2006.

44 Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. Driving with knowledge from the physical world. In *Proc. of the 17th ACM SIGKDD Conf.*, pages 316–324. ACM, 2011.

45 Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th ACM SIGSPATIAL Conference*, pages 99–108. ACM, 2010.