How Packed Is It, Really?

Sariel Har-Peled
Department of Computer Science, University of Illinois, 201 N. Goodwin Avenue, Urbana, IL 61801, USA.

Timothy Zhou
Department of Computer Science, University of Illinois, 201 N. Goodwin Avenue, Urbana, IL 61801, USA

Abstract

The congestion of a curve is a measure of how much it zigzags around locally. More precisely, a curve π is c-packed if the length of the curve lying inside any ball is at most c times the radius of the ball, and its congestion is the minimum c for which π is c-packed. This paper presents a randomized 42-approximation algorithm for computing the congestion of a curve (or any set of segments in the plane). It runs in $O(n \log^2 n)$ time and succeeds with high probability. Although the approximation factor is large, the running time improves over the previous fastest constant approximation algorithm [8], which took $\widetilde{O}(n^{4/3})$ time. We carefully combine new ideas with known techniques to obtain our new near-linear time algorithm.

Keywords and phrases curves, congestion, packedness, Fréchet distance, approximation algorithms

Digital Object Identifier 10.57717/cgt.v4i1.65

Related Version `https://arxiv.org/abs/2105.10776`

Funding *Sariel Har-Peled*: Work on this paper was partially supported by a NSF AF award CCF-1907400 and CCF-2317241.
Timothy Zhou: Work on this paper was partially supported by a NSF AF award CCF-1907400.

Acknowledgements Work on this paper was initiated by discussions of the first author with Anne Driemel. The authors thank the anonymous referees for their detailed and insightful comments that had significantly improved this work.

1. Introduction

In 2010, Driemel *et al.* [6] provided a measurement of how "realistic" a curve is (there are several alternative definitions – see [6] and references therein for details). Formally, a curve π is ***c-packed*** if the total length of π inside any ball is bounded by c times the radius of the ball. The minimum c for which the curve is c-packed is the ***congestion*** of the curve. Intuitively, if a curve has high congestion, then it zigzags back and forth around some locality. We expect that many real-world curves do not behave so pathologically – instead, we expect them to exhibit low congestion. For examples of real world curves, see `https://frechet.xyz`. Naturally, if the curve is a tracking of an entity over long enough time, the congestion might be high (for example, a soccer player movement during a whole game, an airplane path over a month, etc).

Curves with low congestion lend themselves to efficient algorithms. Notably, they can be efficiently approximated by simpler curves which nearly preserve Fréchet distances, and therefore the Fréchet distances between them can be approximated in near-linear time [6]. In general, if the congestion is $\Omega(n)$, then computing the Fréchet distance is more difficult. Indeed, assuming the Strong Exponential Time Hypothesis (SETH), even approximating Fréchet distance within a constant factor requires quadratic time [5]. The decision version of the problem is also conjectured to be 3SUM-hard [2]. Note that proving a direct connection between 3SUM-hardness and SETH is still an open problem [12].

We would like to verify that a given curve is indeed c-packed for a low value of c. Some algorithms for c-packed curves do not require knowing the value of the congestion – rather, their analyses show that if the curve is c-packed for some small c, then the algorithms run in near-linear time. However, verifying that curves are c-packed would increase our confidence that these algorithms are generally applicable. This leads to the question of how quickly can one estimate or compute the congestion. In this paper, we present a constant-factor approximation algorithm for the congestion that runs in near-linear time.

Disks and squares are all the same. As far as the congestion is concerned, whether we use disks/balls or squares/cubes in the definition is the same up to a constant factor (i.e., $\sqrt{2}$ in the plane). Thus, in the following, we work usually in settings of using squares, since it is easier.

Previous work. As mentioned earlier, the concept of c-packedness was introduced in Driemel *et al.* [6]. Computing the congestion exactly runs into the issue of minimizing sums of square roots. As such, an exact algorithm in the standard RAM model is unlikely. The work of Vigneron [11] provided a $(1+\varepsilon)$-approximation algorithm which runs in $O\big((n/\varepsilon)^{d+2}\log^{d+2}(n/\varepsilon)\big)$ time. Gudmundsson *et al.* offered a cubic-time algorithm for this problem [9]. Aghamolaei *et al.* [1] gave a $(2+\varepsilon)$-approximation for the problem of approximating the congestion of planar curves, but unfortunately their running time seems to be at least quadratic.

Some of the previous work [9] was also concerned with computing hotspots, which are small regions in the plane where many segments pass through. Computing regions of high congestion naturally leads to finding hotspots. However, one usually fixes the resolution of the desired hotspots (i.e. the radius of the balls being intersected) before searching them.

More recently, Gudmundsson *et al.* [8] gave a $(6+\varepsilon)$-approximation to the congestion/packedness of a polygonal curve in $\widetilde{O}(n^{4/3}/\varepsilon^4)$ time. As part of their algorithm, they showed that one can quickly find a set of $O(n)$ squares whose congestions yield a constant approximation to the congestion of the curve. They also noted that the problem of finding the congestion of these squares with respect to the segments seems quite similar to the Hopcroft problem, as discussed below.

The Hopcroft problem. Given a set P of n points in the plane, and a set L of n lines in the plane, the question is whether there is a point of P that is incident to one of the lines of L. There is an $\Omega(n^{4/3})$ lower bound for Hopcroft's problem due to Erickson [7]. As usual, this lower bound holds only in a restricted model of computation (algebraic decision tree model), but it is believed to hold in broader models of computation. The belief that this lower bound is correct (up to maybe some polylog noise) in any reasonable model of computation is quite important as it provides matching lower bound to the main results known in range searching.

In our language, the Hopcroft problem can be stated as having a set of n segments, and a set of n (disjoint) squares, and asking for the maximum congestion of the squares in relation to the segments.

The challenge. If we replace a point by a short segment, then deciding the packedness for sufficiently small squares centered at these points is equivalent to solving the Hopcroft problem. Naturally, this reduction of hardness only works if the approximation is quite small (say $<$ 2-approx). For this reason, Gudmunsson *et al.* [8] deemed it unlikely that their approach, "or a similar approach, can lead to a considerably faster algorithm" for computing congestion. Despite this lower bound working only if the approximation constant is small, it is quite interesting to figure out if one can break this “lower bound” (by providing a worse approximation).

Our result. Despite this difficulty, we present a randomized $O(n \log^2 n)$ time algorithm that provides a constant approximation to the congestion of a set of segments in $\mathbb{R}^2$. The algorithm bypasses the barrier presented by Hopcroft’s problem by observing that, when computing congestion, the generated instances for the Hopcroft problem have high congestion. In such scenarios, we do not need to compute the congestion of $\Omega(n)$ disjoint squares exactly (as required by the Hopcroft problem).

Sketch of algorithm. The *congestion* of a square with respect to a curve π is the total length of π inside it, divided by its sidelength. Following [8], we reduce the problem of approximating the congestion of a curve to that of computing the congestion of $O(n)$ squares. Then we build a “few” quadtrees whose cells approximate these squares, so that it suffices to compute the congestion of the quadtrees.

We can compute the congestion of a quadtree cell by finding all the segments of the curve which intersect it, then explicitly computing the total length of the intersections. However, naively searching for all the segments intersecting each of the squares takes $O(n^2)$ time. To speed things up, we store each segment at some quadtree cell with comparable length. For a given cell, its short segments are those stored at descendants in the quadtree, and its long segments are those stored at ancestors. To find the congestion of a cell, we compute the sum of its long and short congestions – the congestions with respect to all of its long and short segments, respectively. We can quickly compute the short congestion of a cell by summing the lengths of short segments in the quadtree bottom-up.

Exactly computing the long congestion is somewhat tricky. Fortunately, approximating the long congestion only requires counting the maximum number of long segments intersecting any cell. If every cell intersects only a few long segments, then we can quickly enumerate all the intersections by searching the quadtree top-down. If there is a cell intersecting many long segments, then the algorithm performs an exponential search to guess how many intersections it has. The algorithm quickly verifies its guess by taking a random sample of input segments, enumerating its intersections with each quadtree cell, and using the counts to estimate the maximum number of intersections.

Highlight. This work “bends” what was previously believed to be a lower bound on the running time for approximating congestion (i.e., $\Omega(n^{4/3})$). While most of the tools we use are standard, the way we combine them is non-trivial and offers new insights into the problem.

Since we lose constant factors in several places, the resulting approximation factor is quite bad compared to previous work (i.e., 42 vs. 6). However, our algorithm runs in near-linear time, which is significantly faster. We find this result surprising, as it bypasses the barrier formed by the Hopcroft problem mentioned above.

Paper organization. We start in Section 2 by providing some definitions and background. In Section 3, we reduce the problem of computing the congestion to that of computing the congestions of a small number of quadtrees. This in turn reduces to the problem of computing the congestion from long and short segments. The short congestion is handled in Section 3.3, while the main challenge of approximating the long congestion is addressed in Section 4, where we approximate the load of the long segments – i.e., the maximum number of long segments intersecting a single cell of the quadtree. In Section 5, we put everything together. We conclude in Section 6 with a few remarks.

2. Preliminaries

2.1. Standard tools

▶ **Definition 1.** *For a real positive number τ, let G_τ be the* **grid** *partitioning the plane into axis-parallel squares of sidelength τ. Formally, this grid is defined by the mapping $\mathsf{G}_\tau(x, y) = \left(\lfloor x/\tau \rfloor, \lfloor y/\tau \rfloor\right)$. The number τ is the* **width** *or* **sidelength** *of G_τ. For integers i, j, the (i, j)-***grid cell** *is the $\tau \times \tau$ square formed by the set $\mathsf{G}_\tau^{-1}(i, j)$.*

▶ **Definition 2.** *A square is a* **canonical square** *if it is contained inside the unit square, it is a cell in a grid G_w, and w is a power of two. That is, the square corresponds to a node in the infinite quadtree defined over $[0, 1)^2$. The grid generating a canonical square is a* **canonical grid**.

2.2. Congestion

▶ **Definition 3.** *Let $\square = \square(p, r)$ denote the axis-parallel square in $\mathbb{R}^2$ centered at a point $p \in \mathbb{R}^2$ with* **sidelength** *$2r$. The square $\square$ can be interpreted as a ball in the L_∞ norm, and as such, its* **radius** *is r.*

▶ **Definition 4.** *For a segment s, let $\|s\|$ denote the length of s. Similarly, for a set of segments $\mathcal{S}$, let $\|\mathcal{S}\| = \sum_{s \in \mathcal{S}} \|s\|$ denote the total length of segments in $\mathcal{S}$.*

For a square $\square = \square(p, r)$, the **conflict list** *of $\square$ (for a set $\mathcal{S}$ of segments) is the set of segments intersecting $\square$, that is*

$$L(\square) = \{s \in \mathcal{S} \mid s \cap \square \neq \emptyset\}.$$

Let

$$\mathcal{S} \sqcap \square = \{s \cap \square \mid s \in L(\square)\}$$

be the clipping of the segments of $L(\square)$ to $\square$. The **congestion** *of the square $\square$, with respect to $\mathcal{S}$, is*

$$\mathsf{c}(\square) = \mathsf{c}_{\mathcal{S}}(\square) = \|\mathcal{S} \sqcap \square\| / r.$$

The **congestion** *of the set of segments $\mathcal{S}$ is $\mathsf{c}(\mathcal{S}) = \max_{p,r} \mathsf{c}_{\mathcal{S}}\left(\square(p, r)\right)$. Given a set of squares Ξ, its* **congestion** *is $\mathsf{c}_{\mathcal{S}}(\Xi) = \max_{\square \in \Xi} \mathsf{c}(\square)$.*

▶ **Definition 5.** *For a constant $c > 0$, a set $\mathcal{S}$ of segments in $\mathbb{R}^2$ is* **c-packed** *if, for any point $p \in \mathbb{R}^2$ and any value $r > 0$, the total length of the segments of $\mathcal{S}$ inside a square $\square = \square(p, r)$ is at most cr. That is, the congestion of $\mathcal{S}$ is at most c.*

Thus, the congestion of $\mathcal{S}$ is the minimum c for which $\mathcal{S}$ is c-packed. We are interested in approximating $\mathsf{c}(\mathcal{S})$. To this end, we follow Gudmundsson *et al.* [8], who reduced the problem to querying the lengths of intersections between the curve and some squares. While Gudmundsson *et al.* state their result for a curve, it holds for any set of segments.

▶ **Lemma 6** (Lemma 12 in [8]). *Given a set $\mathcal{S}$ of n segments in the plane, and a parameter $\varepsilon \in (0, 1)$, one can compute, in $O(n \log n + n/\varepsilon^2)$ time, a set $\mathcal{G}_{\mathcal{S}}$ of $O(n/\varepsilon^2)$ axis-aligned squares, such that $\mathsf{c}(\mathcal{S}) \geq \mathsf{c}_{\mathcal{S}}(\mathcal{G}_{\mathcal{S}}) \geq \mathsf{c}(\mathcal{S})/(6 + \varepsilon)$.*

▶ Remark 7. For completeness, we sketch informally an alternative proof for (a weaker version of) Lemma 6. Let P be the set of endpoints of the segments of $\mathcal{S}$, and construct a (randomly translated) compressed quadtree T for the points of P. For every node v in T, we add the square C_v to the set of candidate squares $\mathcal{G}_{\mathcal{S}}$, and we also add a few scaled copies, say $2C_v$ and $4C_v$ to $\mathcal{G}_{\mathcal{S}}$.

Let $\square$ be the square with maximum congestion $c(\mathcal{S})$. We replace $\square$ by a square $\square' \supseteq \square$ that is centered in one of the endpoints of P, with congestion at least $c(\mathcal{S})/2$ (the interesting case is when $\square$ contains no point of P, but then just enlarge it till it does).

We then continue enlarging $\square'$ till it hits another point of P. Let $\square''$ be the resulting square. Informally, the loss in congestion is a constant. Now, the center of $\square''$ and a point on its boundary both belong to P. With constant probability $\square''$ is fully contained in a cell C_v of a node v of the quadtree (or its enlarged copied added explicitly to $\mathcal{G}_{\mathcal{S}}$), that is only a constant factor bigger. Thus, a square in $\mathcal{G}_{\mathcal{S}}$ provides the desired approximation to the congestion.

3. The algorithm: The long and short of it

3.1. Reduction to quadtrees

In what follows, let $\mathcal{G}_{\mathcal{S}}$ be the set of squares computed by Lemma 6 for $\mathcal{S}$ (the value of ε would be specified shortly). To approximate $c(\mathcal{S})$, it suffices to approximate $c_{\mathcal{G}_{\mathcal{S}}}(\mathcal{S})$.

Since congestion is invariant under translation and scaling, we might as well assume that $\mathcal{S}, \mathcal{G}_{\mathcal{S}} \subseteq [0, 1/8]^2$. We can randomly scale both sets by a random number $\mathsf{s}_\mathsf{r} \in [1, 2]$, and shift both sets by a random vector $\mathsf{u}_\mathsf{r} \in [0, 1/2]^2$. Let $\Lambda(p) = \mathsf{s}_\mathsf{r} p + \mathsf{u}_\mathsf{r}$ be the resulting affine mapping. We compute for each square $\square \in \Lambda(\mathcal{G}_{\mathcal{S}})$ the smallest canonical square $\textsc{canon}(\square)$ that contains it – conceptually, consider the infinite quadtree, and computing the lowest node v in the quadtree that its cell contains $\square$ – the square C_v is $\textsc{canon}(\square)$. Algorithmically, $\textsc{canon}(\square)$ can be computed in $O(1)$ time using the floor function and some basic bit operations, see [10]. Next, we build a quadtree that has all these marked canonical squares as nodes. This can be done in $O(n \log n)$ time [10]. The idea is to repeat this process sufficient number of times, such that in one of the generated quadtrees, $\textsc{canon}(\square)$ and $\square$ are almost identical, and this holds for all the squares of interest.

▶ **Lemma 8.** *For a square $\widehat{\square} \in \mathcal{G}_{\mathcal{S}}$, and a parameter $\varepsilon \in (0, 1/2)$, consider its randomly scaled and shifted copy $\square = \mathsf{s}_\mathsf{r}\widehat{\square} + \mathsf{u}_\mathsf{r}$, and its canonized version $\square' = \textsc{canon}(\square)$.*

The probability that $r = \mathsf{r}(\square) \leq \mathsf{r}(\square') \leq (1+\varepsilon)r$ is $\geq (\varepsilon/8)^3$.

Proof. By construction $\square \subseteq \square'$. Let $\ell = \mathsf{r}(\widehat{\square})$, and let $L = 2^{\lceil \log_2 \ell \rceil}$ be the rounding up of ℓ to its closest power of 2. For simplicity of exposition assume that $\zeta = L/l > (1+\varepsilon)$ – as otherwise, one can apply the analysis to $2L/\ell$.

The optimal scaling for our purposes is ζ, but let us be slightly less greedy, and consider the random scaling s_r to be *good* if $\mathsf{s}_\mathsf{r} \in [(1-\varepsilon/4)\zeta, (1-\varepsilon/8)\zeta]$. This interval is of length $(\varepsilon/8)\zeta \geq \varepsilon/8$. Thus, the random scaling is good with probability $\geq \varepsilon/8$, and assume this happened.

Consider the canonical grid G_{2L}, and $r = \mathsf{r}(\square)$. We have that $2r \in [(1-\varepsilon/4)2L, (1-\varepsilon/8)2L]$. as such, the set of all good translations

$$U = \left\{ (x, y) \in \mathbb{R}^2 \;\middle|\; (x, y) + \mathsf{s}_\mathsf{r}\widehat{\square} \text{ contained in a cell of } \mathsf{G}_{2L} \right\}$$

is a grid like set with sidelength $2L$, where each grid point is replaced by a square of sidelength $\geq (\varepsilon/8)2L$. The probability that the random shifting u_r falls in U is at least $(\varepsilon/8)^2$.

If both things happen, then $r \leq L \leq (1+\varepsilon)(1-\varepsilon/4)L \leq (1+\varepsilon)r$. Namely, $\square \subseteq \square'$, and $r = \mathsf{r}(\square) \leq \mathsf{r}(\square') \leq (1+\varepsilon)r$, as desired. ◀

▶ **Definition 9.** *For a compressed quadtree $\mathcal{T}$, let $\Xi(\mathcal{T})$ be the set of all squares formed by nodes of $\mathcal{T}$. The* **congestion** *of a quadtree $\mathcal{T}$ for a set of segments $\mathcal{S}$ is the quantity $c_{\mathcal{S}}(\mathcal{T}) = c_{\mathcal{S}}(\Xi(\mathcal{T}))$.*

▶ **Lemma 10.** *Given a set $\mathcal{S}$ of n segments, one can compute, in $O(n \log^2 n)$ time, $m = O(\log n)$ (shifted) compressed quadtrees, $\mathcal{T}_1, \ldots, \mathcal{T}_m$, such that $\frac{1}{7} c(\mathcal{S}) \leq \max_i c_{\mathcal{S}}(\mathcal{T}_i) \leq c(\mathcal{S})$. This holds with high probability.*

Proof. Let $\varepsilon \in (0,1)$ be a sufficiently small constant to be specified shortly. Compute the set $\mathcal{G}_{\mathcal{S}}$ of squares specified by Lemma 6. Next, compute $m = O(\varepsilon^{-3} \log n)$ randomly shifted copies $\mathcal{G}^1_{\mathcal{S}}, \ldots, \mathcal{G}^m_{\mathcal{S}}$ of $\mathcal{G}_{\mathcal{S}}$, as described above. For each one of them we compute a compressed quadtree. Each such compressed quadtree, can be interpreted as a shifted compressed quadtree over the original set of squares $\mathcal{G}_{\mathcal{S}}$. And let $\mathcal{T}_1, \ldots, \mathcal{T}_m$ be these quadtrees.

Fix a square $\square \in \mathcal{G}_{\mathcal{S}}$. By Lemma 8, with probability $p \geq (\varepsilon/8)^3$, there is a node u, and thus its cell $\square'$, such that $\square \subseteq \square'$ and $\mathsf{r}(\square') \leq (1+\varepsilon)\mathsf{r}(\square)$. Thus, we have $c_{\mathcal{S}}(\square') \geq c_{\mathcal{S}}(\square)/(1+\varepsilon)$.

In particular, the probability that this "good" containment does not happen for $\square$, in all m quadtrees, is $(1-p)^m \leq 1/n^{O(1)}$. We conclude that, high probability, for all squares $\square \in \mathcal{G}_{\mathcal{S}}$, there is at least one node/square, in the computed quadtrees, that tightly contains $\square$.

This readily implies that

$$\max_i c_{\mathcal{S}}(\mathcal{T}_i) \geq \frac{1}{1+\varepsilon} \max_{\square \in \mathcal{G}_{\mathcal{S}}} c_{\mathcal{S}}(\square) \geq \frac{1}{(1+\varepsilon)(6+\varepsilon)} c(\mathcal{S}) \geq \frac{1}{7} c(\mathcal{S}),$$

for $\varepsilon = 1/10$. ◀

▶ Remark 11. Throughout the algorithm, we use compressed quadtrees, rather than regular quadtrees. The use of compressed quadtrees is necessary to get an efficient runtime, but it does not affect the description of our algorithm significantly. (Intuitively, the compressed quadtree compresses paths and not cells, so one can easily ensure that all cells of interest appear in the compressed quadtree as regular cells). As such, from this point on, we use quadtree as a shorthand for a compressed quadtree, and we ignore the minor low-level technical details that arise because of the compression.

Specifically, the only issue in the algorithm where this would make a difference is in traversing down the tree and assuming that the child is exactly half the size of its parent. This can easily be fixed by working directly with the node own record of its dimensions.

3.2. First steps towards approximating the congestion of a quadtree

Given a set $\mathcal{S}$ of n segments and a quadtree $\mathcal{T}$ of size $O(n)$, we wish to compute the congestion of $\mathcal{T}$. To simplify the exposition, we assume that any shift needed is applied to the input segments, and the quadtree is thus standard one built over $[0,1]^2$.

3.2.1. A naïve exact algorithm for the congestion of a quadtree

At each node $v \in \mathcal{T}$, the algorithm stores a ***conflict list*** $L(v)$ – a list of all the segments of $\mathcal{S}$ intersecting the cell $\square_v$ associated with v. It begins by storing $\mathcal{S}$ at the root of $\mathcal{T}$, then recursively traverses down the tree. At each parent node u, the algorithm sends the list $L(u)$ to all its children. Each child v finds all the segments which intersect its cell $\square_v$ and adds

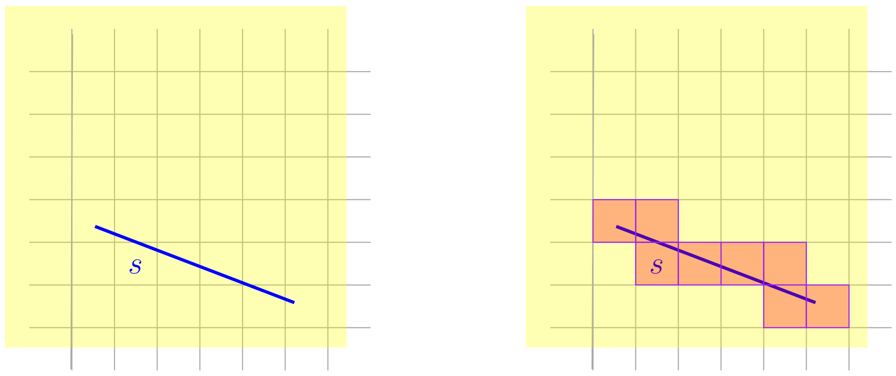

Figure 3.1 Computing a maximal set of α-long segments for a segment s (see Lemma 13). In this case, the segment has length 5, and each grid cell is of radius 0.5, so the segment is 10-long for all the cells of the grid it intersects.

them to its own list. At the end of this process, the algorithm has computed the segments intersecting each quadtree node, and it can use them to compute the congestion. Overall, this algorithm takes $O(n^2)$ time.

3.2.2. The long and the short of it

To speed up the naïve algorithm, we implement several strategies. The first is to register the segments directly in the cells at a suitable resolution of the quadtree.

▶ **Definition 12** (long/short threshold). *Let $\alpha > 0$ be a fixed integer constant, to be specified shortly*[1]*. The parameter α is the* **transition** *constant. A segment s is* **α-long** *(resp.,* **α-short***) for a square $\square = \square(p, r)$ if it intersects the interior of $\square$ and $\|s\| \geq \alpha r$ (resp., $\|s\| < \alpha r$). For a square $\square$ with radius r, its set of α-long (resp., α-short) segments is denoted by $\mathcal{S}_{\geq\alpha}(\square)$ (resp., $\mathcal{S}_{<\alpha}(\square)$).*

▶ **Lemma 13.** *For each segment $s \in \mathcal{S}$, let $\mathsf{G}(s, \alpha)$ be the set of interior-disjoint canonical squares of maximal size for which s is α-long. There are at most $O(1 + \alpha)$ such squares, and they can be computed in $O(1 + \alpha)$ time.*

Proof. Let $i = \lfloor \log_2(\|s\| / \alpha) \rfloor$, and $\tau = 2^i$. Since

$$\alpha\tau = \alpha 2^{\lfloor \log_2(\|s\|/\alpha) \rfloor} \leq \alpha \cdot \frac{\|s\|}{\alpha} \leq \|s\| < \alpha 2^{1+\lfloor \log_2(\|s\|/\alpha) \rfloor} < 2\alpha\tau,$$

the segment s intersects at most $2(\alpha + 1)$ horizontal lines of the grid G_τ. This implies that s can intersect at most $4(\alpha + 1) + 1$ grid cells of G_τ. Computing the grid cells that intersect s is a classical problem in graphics (i.e., line drawing problem) which can be solved in $O(1 + \alpha)$ time; see Figure 3.1. ◀

[1] Spoiler alert! The butler did it, $\alpha = 20$, and the hero dies in the end. The hero did try to expose α, but it was too late for them.

3.2.3. Registering the segments

The refined quadtree $\mathcal{T}^+$. Given the set $\mathcal{S}$ of n segments and the above quadtree $\mathcal{T}$, the algorithm first computes the set of canonical squares $\Xi = \mathrm{cells}(\mathcal{T}) \cup \bigcup_{s\in\mathcal{S}} \mathsf{G}(s,\alpha)$; see Lemma 13. The algorithm then computes the (compressed) quadtree $\mathcal{T}^+$ for Ξ. This can be done in $O(n \log n)$ time [10], as $|\mathcal{S}| = n$, $|\mathrm{cells}(\mathcal{T})| = O(n)$, and $|\Xi| = O((1+\alpha)n) = O(n)$, as α is a constant. Each square of Ξ is now present as a cell of a node of the computed quadtree.

The algorithm now stores every segment $s \in \mathcal{S}$ in the cells of $\mathsf{G}(s,\alpha)$. Each cell $\square \in \mathsf{G}(s,\alpha)$ corresponds to a node v in the quadtree, and the algorithm stores s in $L(v)$. This takes $O(n \log n)$ time; see [10]. Thus, for every quadtree node v, the algorithm computes a list $L_{\mathrm{long}}(v)$ of segments registered there. Segments registered in this list are long for the cell $\square_v$ but short for cells in higher levels of the quadtree.

Propagating each segment up to the parent node, we also register each segment in a list $L_{\mathrm{short}}(v)$ of another node v (this is done only for one level up). Segments registered in this list are short for the cell $\square_v$ but long for cells in lower levels of the quadtree. Since computing the registering cells for each segment takes $O(1)$ time, computing the lists $L_{\mathrm{short}}(\cdot)$ and $L_{\mathrm{long}}(\cdot)$ for all nodes in the tree takes $O(n)$ time, and the lists themselves have total length $O(n)$.

A segment is registered only once as a short or long segment on any path in the quadtree.

▶ **Definition 14.** *The* **α-long congestion** *of $\square$ is $\cancel{c}_{\geq\alpha}(\square) = \|\mathcal{S}_{\geq\alpha}(\square) \sqcap \square\| / r$. Similarly the* **$\alpha$-short congestion** *of $\square$ is $\cancel{c}_{<\alpha}(\square) = \|\mathcal{S}_{<\alpha}(\square) \sqcap \square\| / r$. Given a quadtree $\mathcal{T}$, its* **α-long congestion** *and* **α-short congestion** *are*

$$\cancel{c}_{\geq\alpha} = \cancel{c}_{\geq\alpha}(\mathcal{T}) = \max_{\square\in\mathrm{cells}(\mathcal{T})} \cancel{c}_{\geq\alpha}(\square) \qquad \textit{and} \qquad \cancel{c}_{<\alpha} = \cancel{c}_{<\alpha}(\mathcal{T}) = \max_{\square\in\mathrm{cells}(\mathcal{T})} \cancel{c}_{<\alpha}(\square).$$

For a node $v \in \mathcal{T}^+$, let $\mathrm{anc}(v)$ (resp., $\mathrm{desc}(v)$) be the list of ancestors (resp., descendants) of v in the tree $\mathcal{T}^+$. Here (emptily) $v \in \mathrm{anc}(v)$ and $v \in \mathrm{desc}(v)$. Consider a node $v \in \mathcal{T}^+$, and let $\square_v$ be its associated square. We have that

$$\mathcal{S}_{\geq\alpha}(\square_v) = \bigcup_{u\in\mathrm{anc}(v)} (L_{\mathrm{long}}(u) \cap \square_v) \qquad \text{and} \qquad \mathcal{S}_{<\alpha}(\square_v) = \bigcup_{u\in\mathrm{desc}(v)} L_{\mathrm{short}}(u).$$

To summarize, we have described how to augment a quadtree $\mathcal{T}$ by adding more cells and registering short and long segments at the cells, yielding a new quadtree $\mathcal{T}^+$. In what follows, we use this stored information to compute the long and short congestions of $\mathcal{T}^+$ (which are at least the congestions of the sub-quadtree $\mathcal{T}$).

3.3. Computing the congestion of the short segments

The α-short congestion is computed using dynamic programming, as described next.

▶ **Lemma 15.** *Given a set $\mathcal{S}$ of n segments in the plane and a quadtree $\mathcal{T}^+$ of size $O(n)$, one can compute, in $O(n)$ time, the α-short congestion for all the nodes of $\mathcal{T}^+$, where α is a constant.*

Proof. We compute the α-short congestion of a quadtree via dynamic programming. The algorithm finds the total length of short segments intersecting each leaf and propagates the values upward.

For every node $v \in \mathcal{T}^+$, the algorithm computes the quantity $\|L_{\mathrm{short}}(v) \sqcap \square_v\|$. Computing the value for node v requires time proportional to the total size of the list $L_{\mathrm{short}}(v)$, so doing it for all the nodes of the tree takes $O(n)$ time overall. Next, the algorithm traverses the tree bottom-up. For each node along the way, it computes the total lengths of the intersecting short segments:

$$\Delta_v = \sum_{u \in \mathrm{desc}(v)} \left\| L_{\mathrm{short}}(u) \sqcap \square_u \right\| = \left\| L_{\mathrm{short}}(v) \sqcap \square_v \right\| + \sum_{u \text{ child of } v} \Delta_u.$$

It is easy to verify that $\Delta_v = \left\| \mathcal{S}_{<\alpha}(\square_v) \sqcap \square_v \right\|$. The lemma follows. ◀

4. Approximating the maximum load of the long segments in a quadtree

Handling the long congestion quickly seems challenging. Instead, here we would quickly approximate a proxy for this quantity – the maximum number of such segments visiting a single node.

▶ **Definition 16.** *The* α-**load** *of a quadtree* $\mathcal{T}^+$ *is the quantity* $\rho = \max_{\square \in \mathcal{T}^+} |\mathcal{S}_{\geq\alpha}(\square)|$.

4.1. A naïve algorithm for computing the α-load

To compute its exact load of a given square, we need to find the long segments intersecting the cell. We can compute the conflict list for each cell by pushing long segments downward from the conflict lists of its ancestors (as done in the naïve algorithm of Section 3.2.1). Unfortunately, these lists can get quite long.

▶ **Lemma 17.** *One can compute the* α*-load congestion of* $\mathcal{T}^+$ *in* $O(n \log n + \rho n)$ *time. More generally, given a set* $R \subseteq \mathcal{S}$ *and a threshold* t*, one can decide whether* $\rho(R) = \max_{\square \in \mathcal{T}^+} |R \cap \mathcal{S}_{\geq\alpha}(\square)| \leq t$*, in* $O(n \log n + tn)$ *time.*

Proof. The algorithm starts with the precomputed lists $L_{\mathrm{long}}(u)$ and traverses the tree top-down. At each node, it pushes the stored list down to the children. Each child v selects the segments of the incoming list that intersect its cell, and takes the union of this filtered incoming list with $L_{\mathrm{long}}(v)$. This yields the conflict list of v of *all* α-long segments that intersect it, denoted by $L_{\mathrm{LONG}}(v)$. It then pushes this list down to its children, and so on.

For each node v of the quadtree, it is now straightforward to compute the congestion of the segments of $L_{\mathrm{LONG}}(v)$ (or the length of the list $L_{\mathrm{LONG}}(v)$) for the cell of the node v. It follows that this computes the α-long congestion for each node of the quadtree. Since the maximum length of the lists sent down is ρ, the claim follows.

The second algorithm works in a similar fashion, except that the algorithm propagates downwards only segments of R. If in any point in time the computed conflict list gets bigger than t, the algorithm bails out, returning that $\rho(R) > t$. ◀

4.2. From long congestion to load, and back

Instead of computing the α-long congestion of a quadtree exactly, we content ourselves with a constant-factor approximation, by using the load instead.

▶ **Lemma 18.** *For a cell* $\square = \square(p, r)$ *of* $\mathcal{T}^+$*, we have* $\dfrac{\mathcal{C}_{\geq\alpha}(\square)}{\sqrt{8}} \leq |\mathcal{S}_{\geq\alpha}(\square)| \leq \dfrac{1+\alpha}{\alpha}\mathcal{C}(\mathcal{S})$.

Proof. The intersection of each segment of $\mathcal{S}_{\geq\alpha}(\square)$ with $\square$ can have length at most $\sqrt{2} \cdot 2r = \sqrt{8}r$ (as the sidelength of $\square$ is $2r$), so $\mathcal{C}_{\geq\alpha}(\square) = \left\| \square \sqcap \mathcal{S}_{\geq\alpha}(\square) \right\| / r$, and $\left\| \square \sqcap \mathcal{S}_{\geq\alpha}(\square) \right\| \leq |\mathcal{S}_{\geq\alpha}(\square)| \cdot r\sqrt{8}$, and the first inequality follows.

As for the second inequality, consider $\square' = \square(p, (1+\alpha)r)$. By definition, each long segment $s \in \mathcal{S}_{\geq\alpha}(\square)$ has length at least αr and intersects $\square$. The length of $s \cap \square'$ is at least αr, see Figure 4.1. As such, we have

$$|\mathcal{S}_{\geq\alpha}(\square)| \cdot \frac{\alpha r}{(1+\alpha)r} \leq \frac{\|\mathcal{S}_{\geq\alpha}(\square) \sqcap \square'\|}{(1+\alpha)r} \leq c(\square') \leq c(\mathcal{S}). \qquad \blacktriangleleft$$

4.3. Exponential search for maximum α-load

It remains to estimate the load (i.e., maximum number of α-long segments intersecting any cell in the quadtree $\mathcal{T}^+$). The basic idea is to couple random sampling together with exponential search, to approximate the maximum load. If during a round an "overflow" occurs, the algorithm increases the guess for the load by (say a factor of 2), and moves on to the next round. Using sampling and exponential search to estimate a quantity is an old idea. In the context of geometric settings, it was used before to estimate the maximum depth of nicely behaved regions, see [3].

4.3.1. The algorithm

Our purpose here is to approximate the α-load of $\mathcal{T}^+$, denoted by ρ, see Definition 16.

Estimation via sampling. We perform an exponential search for the size of the largest conflict list. We use the following standard sampling lemma. It follows by a standard application of Chernoff's inequality – see [4, Lemma 2.7] for a proof.

▶ **Lemma 19.** *Consider two (finite) sets $B \subseteq \mathcal{S}$, where $n = |\mathcal{S}|$. Let $\xi \in (0,1)$ and $\varphi \in (0,1/2)$ be parameters, and let $r = \left\lceil c_2 \xi^{-2} \frac{n}{g} \log \varphi^{-1} \right\rceil$, where c_2 is a sufficiently large constant.*

Let $g > 0$ be a user-provided guess for the size of $|B|$. Consider a random sample R, taken from $\mathcal{S}$ by picking each element with probability $\frac{r}{n}$, Next, consider the estimate $Y = \frac{n}{r}|R \cap B|$ to $|B|$. Then, we have the following:

(A) If $Y < g/2$, then $|B| < g$,

(B) If $Y \geq g/2$, then $(1-\xi)Y \leq |B| \leq (1+\xi)Y$.

Both statements above hold with probability $\geq 1 - \varphi$.

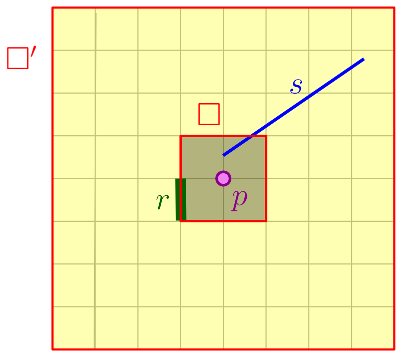

Figure 4.1 An α-long segment for a square $\square = \square(p, r)$ intersects the square $\square' = \square(p, (1+\alpha)r)$ with a segment of length at least αr (here, $\alpha = 3$).

The algorithm. The number of cells in $\mathcal{T}^+$ is $O(n)$, and let $\varphi = 1/n^{O(1)}$. Let $\varepsilon \in (0,1)$ be a prespecified approximation parameter, and let $\xi = \varepsilon/3$. At each round, the algorithm has a threshold number g_i and checks whether (i) $\rho > 8g_i$ (roughly), or (ii) $\rho = \Theta(g_i)$ and it can approximated reliably and quickly. The ith round start, with parameters

$$g_i = 2^{i-1} \qquad \text{and} \qquad r_i = \lceil c_2 \xi^{-2} \tfrac{n}{g_i} \log \varphi^{-1} \rceil \,. \tag{4.1}$$

where c_2 is a sufficiently large constant. If the ith round failed, the algorithm goes on to the next round (i.e., by increasing i by one).

Let I be the first integer such that $r_i < n$ (i.e., $I = \Theta(\delta^{-1} \log n)$), and the algorithm starts with $i = I$. This first round, the algorithm uses Lemma 17 directly. If it finds some conflict list (for $\mathcal{S}$) that contains more than $8g_i = 8g_I$ segments at any point during the execution, it concludes that g_i is too small. Thus this round is be a failure, and it goes on to the next round. Otherwise, the algorithm computes ρ exactly and returns its value.

For later rounds, and larger values of g_i, the round begins by taking a random sample $R_i \subseteq \mathcal{S}$ of the segments. Each segment is included in R_i independently with probability $\varsigma_i = r_i/n$ (i.e., a random sample according to Lemma 19). The algorithm then calls the subroutine of Lemma 17 with R_i as the list of segments and with $U_i = 8g_i\varsigma_i$ as the threshold. If the output reveals that $\rho(R) > U_i$, then the guess g_i is too small, the round failed, and the algorithm continues to the next round.

If a round succeeds (and $i \neq I$), the algorithm outputs $(1-\xi)Y$ for $Y = \frac{n}{r_i}\rho(R)$ as an estimate from below to ρ.

4.3.2. Analysis

Running time. The first round takes $O(n \log^2 n)$ time. Since there are only n segments in $\mathcal{S}$, the algorithm always stops by round $O(\log n)$. The threshold used in each round is

$$U_i = 8g_i\varsigma_i = \frac{8g_i r_i}{n} = O\Big(\frac{g_i n}{n\xi^2 g_i n} \log n^{O(1)}\Big) = O\Big(\frac{\log n}{\varepsilon^2}\Big).$$

As such, the running time of each round is $O(\varepsilon^{-2} n \log n)$; see Lemma 17, and the overall running time is $O(\varepsilon^{-2} n \log^2 n)$.

The result.

▶ **Lemma 20.** *Using the above randomized algorithm, one can compute, in $O(\varepsilon^{-2} n \log^2 n)$ time, a number Δ, such that with high probability, we have $\Delta \leq \rho \leq (1+\varepsilon)\Delta$, where ρ is the α-load of $\mathcal{T}^+$, see Definition 16.*

Proof. If the algorithm terminated after the first round, then it output ρ exactly, and we are done.

Otherwise, if it failed in a round, it must be that it found a node that its conflict list exceeds the "expected" threshold $U_i = 8g_i\varsigma_i$. Lemma 19 implies that the random sample in such cases estimates the size of the original conflict list correctly, and it is at least of size $(1-\xi)U_i > 4g_i$. Namely, g_i is (way) too small, and this decision was made correctly with probability $1 - 1/n^{O(1)}$.

If the algorithm succeeded in a round, then Lemma 19 implies that $\rho \leq (1+\xi)U_i < 16g_i$. Since the algorithm failed the previous round, we have that $\rho > 4g_{i-1} \geq 2g_i$. Lemma 19 then implies that ρ is $(1 \pm \xi)$-estimated correctly. Specifically, for the jth "heavy" node (i.e., conflict list size ρ_j is larger than g_i) in the tree, let Y_j be the estimate of its conflict list size.

We have that $(1-\xi)Y_j \leq \rho_j \leq (1+\xi)Y_j$. This inequality clearly holds also on the max value, which implies that

$$(1-\xi)Y \leq \rho \leq (1+\xi)Y \implies Y \leq \rho \leq \frac{1+\xi}{1-\xi}Y \leq (1+3\xi)Y = (1+\varepsilon)Y.$$

An important technicality here is that any of the conflict lists in the tree of size smaller than $g_i/2$ are too small after the sampling to compete with the conflict list realizing ρ. Similarly, this also holds for conflict lists of size $\leq g_i$. Thus, all the conflict lists in play for realizing the maximum of the sample are estimated correctly[2]. ◀

5. Approximating the maximum congestion

We seem to have lost our way, so lets try to get back on track. Consider the following quantity

$$\mathcal{C}_{\geq\alpha}(\mathcal{T}^+) = \max_{\square\in\text{cells}(\mathcal{T}^+)} \max\bigl[c_{\geq\alpha}(\square), c_{\mathcal{S}_{\geq\alpha}(\square)}\bigl((1+\alpha)\square\bigr)\bigr],$$

which is the ***augmented*** long congestion of $\mathcal{T}^+$. We define[3]

$$\mathcal{C}_{\mathcal{S}}(\mathcal{T}^+) = \max\bigl(c_{<\alpha}(\mathcal{T}^+), \mathcal{C}_{\geq\alpha}(\mathcal{T}^+)\bigr).$$

Clearly, $c_{\mathcal{S}}(\mathcal{T}^+)/2 \leq \mathcal{C}_{\mathcal{S}}(\mathcal{T}^+) \leq c(\mathcal{S})$.

▶ **Lemma 21.** *We have* $\frac{\alpha}{1+\alpha}\rho \leq \mathcal{C}_{\geq\alpha}(\mathcal{T}^+) \leq \sqrt{8}\rho$.

Proof. The upper bound is immediate. The lower bound readily follows from the argument used in the proof of Lemma 18. ◀

▶ **Lemma 22.** *One can compute, in $O(n\log^2 n)$ time, a quantity $\mathcal{A}_{\mathcal{S}}(\mathcal{T}^+)$, such that $\mathcal{A}_{\mathcal{S}}(\mathcal{T}^+) \leq \mathcal{C}_{\mathcal{S}}(\mathcal{T}^+) \leq 3\mathcal{A}_{\mathcal{S}}(\mathcal{T}^+)$. This holds with high probability.*

Proof. let $\varepsilon = 1/100$, and let Δ be the approximation of ρ computed by the algorithm of Lemma 20. This takes $O(n\log^2 n)$ time. We now compute $c_{<\alpha}(\mathcal{T}^+)$ in $O(n\log n)$ time using the algorithm of Lemma 15. We compute the quantity

$$\mathcal{A}_{\mathcal{S}}(\mathcal{T}^+) = \max\Bigl(c_{<\alpha}(\mathcal{T}^+), \frac{\alpha}{1+\alpha}\Delta\Bigr).$$

Clearly, $\mathcal{A}_{\mathcal{S}}(\mathcal{T}^+) \leq \mathcal{C}_{\mathcal{S}}(\mathcal{T}^+)$. As for the other direction, observe that

$$\frac{\sqrt{8}(1+\varepsilon)(1+\alpha)}{\alpha}\mathcal{A}_{\mathcal{S}}(\mathcal{T}^+) \geq \frac{\alpha}{1+\alpha}\Delta \cdot \frac{\sqrt{8}(1+\varepsilon)(1+\alpha)}{\alpha} \geq \sqrt{8}\rho \geq \mathcal{C}_{\geq\alpha}(\mathcal{T}^+),$$

by Lemma 21. Numerical calculations shows that for $\varepsilon = 1/100$ and $\alpha = 20$, $\frac{\sqrt{8}(1+\varepsilon)(1+\alpha)}{\alpha} \leq 3$, which establish the claim. ◀

Finally, we arrive at our main result.

[2] So, confusingly, while the winning estimate might not be from the node with the largest conflict list, it comes from a node with a conflict list of size very close to it – if it was much smaller, it would not have been able to beat it.

[3] Spoiler alert! The hero is still dead, and it turns out that for $\mathcal{T}_1, \ldots, \mathcal{T}_m$ the quadtrees of Lemma 10, $\max_i \mathcal{C}_{\geq\alpha}(\mathcal{T}_i)$ is a good approximation to $c(\mathcal{S})$.

▶ **Theorem 23.** *Let $\mathcal{S}$ be a set of n segments in the plane. One can compute, in $O(n \log^3 n)$ time, a 42-approximation to $\mathcal{c}(\mathcal{S})$. The algorithm is randomized and succeeds with high probability.*

Proof. Compute, in $O(n \log^2 n)$ time, the $m = O(\log n)$ quadtrees $\mathcal{T}_1, \ldots, \mathcal{T}_m$ of Lemma 10. Each quadtree $\mathcal{T}_i$ is refined as described in Section 3.2.3, into a quadtree $\mathcal{T}_i^+$, and we compute for each one of them the quantity $\zeta_i = \mathcal{A}_{\mathcal{S}}(\mathcal{T}_i^+)$, using the algorithm of Lemma 22 in $O(n \log^2 n)$ time. Overall, the running time is $O(n \log^3 n)$, and all the steps so far succeeded with high probability. The algorithm returns $\zeta = \max_i \zeta_i \leq \mathcal{c}(\mathcal{S})$ as the desired approximation to the congestion. For the quality of approximation, observe that

$$42\zeta = 14 \max_i 3\zeta_i \geq 14 \max_i \mathcal{C}_{\mathcal{S}}(\mathcal{T}_i^+) \geq 7 \max_i \mathcal{c}_{\mathcal{S}}(\mathcal{T}_i^+) \geq 7 \cdot \frac{1}{7} \mathcal{c}(\mathcal{S}) = \mathcal{c}(\mathcal{S}). \qquad ◀$$

6. Conclusions

We provided a near-linear time algorithm that computes a constant factor approximation to the congestion of a polygonal curve (i.e., the minimum c such that the curve is c-packed). We consider the result to be quite surprising, even though the constant is undesirably large (i.e., 42).

The new algorithm works verbatim in any constant dimension – our algorithm has not used planarity in any way. The quality of approximation deteriorates with the dimension d, but it is still a constant when d is a constant. The running time remains $O(n \log^3 n)$.

Another important property of the new algorithm is that it does not require the input segments to form a curve. It is natural to conjecture that in the plane, if the input is a polygon curve that does not self intersect, then one should be able to $(1 + \varepsilon)$-approximate the congestion in near linear time. We leave this as an open problem for further research.

As mentioned above, the constant in the approximation quality of the new algorithm is not pretty (currently 42). Reducing the constant further while keeping the running time near-linear is an interesting problem for future research.

Nothing in our algorithm is a no-starter from a practical point of view. Performing an experimental study using this algorithm is an interesting future research.

References

1 Sepideh Aghamolaei, Vahideh Keikha, Mohammad Ghodsi, and Ali Mohades. Windowing queries using Minkowski sum and their extension to mapreduce. *J. Supercomput.*, April 2020. doi:10.1007/s11227-020-03299-7.

2 Helmut Alt. *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, chapter The Computational Geometry of Comparing Shapes, page 235–248. Springer-Verlag, 2009.

3 B. Aronov and S. Har-Peled. On approximating the depth and related problems. *SIAM J. Comput.*, 38(3):899–921, 2008. URL: http://dx.doi.org/10.1137/060669474, doi:10.1137/060669474.

4 Paul Beame, Sariel Har-Peled, Sivaramakrishnan Natarajan Ramamoorthy, Cyrus Rashtchian, and Makrand Sinha. Edge estimation with independent set oracles. *ACM Trans. Algo.*, 16(4), September 2020. doi:10.1145/3404867.

5 K. Bringmann. Why walking the dog takes time: Frechet distance has no strongly subquadratic algorithms unless SETH fails. In *Proc. 55th Annu. IEEE Sympos. Found. Comput. Sci.* (FOCS), pages 661–670, 2014. doi:10.1109/FOCS.2014.76.

6 A. Driemel, S. Har-Peled, and C. Wenk. Approximating the Fréchet distance for realistic curves in near linear time. *Disc. Comput. Geom.*, 48:94–127, 2012. doi:10.1007/s00454-012-9402-z.

7 Jeff Erickson. On the relative complexities of some geometric problems. *Proc. 7th Canadian Conference on Computational Geometry*, page 85–90, 1995. URL: http://www.cccg.ca/proceedings/1995/cccg1995_0014.pdf.

8 Joachim Gudmundsson, Yuan Sha, and Sampson Wong. Approximating the packedness of polygonal curves. *CoRR*, abs/2009.07789, 2020. to appear in ISAAC 2020. URL: https://arxiv.org/abs/2009.07789, arXiv:2009.07789.

9 Joachim Gudmundsson, Marc van Kreveld, and Frank Staals. Algorithms for hotspot computation on trajectory data. In *Proc. 21st ACM SIG. Int. Conf. Adv. Geo. Info. Sys. (SIGSPATIAL)*, SIGSPATIAL'13, page 134–143, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2525314.2525359.

10 S. Har-Peled. *Geometric Approximation Algorithms*, volume 173 of *Math. Surveys & Monographs.* Amer. Math. Soc., Boston, MA, USA, 2011. URL: http://sarielhp.org/book/, doi:10.1090/surv/173.

11 Antoine Vigneron. Geometric optimization and sums of algebraic functions. *ACM Trans. Algo.*, 10(1), January 2014. doi:10.1145/2532647.

12 Virginia Vassilevska Williams. Hardness of easy problems: Basing hardness on popular conjectures such as the strong exponential time hypothesis (invited talk). In Thore Husfeldt and Iyad A. Kanj, editors, *10th Int. Symp. Param. Exact Comput, IPEC*, volume 43 of *LIPIcs*, pages 17–29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.IPEC.2015.17.