Computing the Stretch Factor of Paths, Trees, and Cycles in Weighted Fixed Orientation Metrics

Christian Wulff-Nilsen*

Abstract

Let G be a connected graph with n vertices embedded in a metric space with metric δ. The stretch factor of G is the maximum over all pairs of distinct vertices $u, v \in G$ of the ratio $\delta_G(u,v)/\delta(u,v)$, where $\delta_G(u,v)$ is the metric distance in G between u and v. We consider the plane equipped with a weighted fixed orientation metric, i.e. a metric that measures the distance between a pair of points as the length of a shortest path between them using only a given set of $\sigma \geq 2$ weighted fixed orientations. We show how to compute the stretch factor of G in $O(\sigma n \log^2 n)$ time when G is a path and in $O(\sigma n \log^3 n)$ time when G is a tree or a cycle. For the L_1-metric, we generalize the algorithms to d-dimensional space and show that the stretch factor can be computed in $O(n \log^d n)$ time when G is a path and in $O(n \log^{d+1} n)$ time when G is a tree or a cycle. All algorithms have $O(n)$ space requirement. Time and space bounds are worst-case bounds.

1 Introduction

Designing modern microchips is a complicated process involving several steps. One of these steps, the so called routing step, deals with the problem of finding a layout of wires on the chip interconnecting a given set of pins.

Many factors need to be taken into consideration when finding such a layout. An important measure is the total wire length. Minimizing this will help reduce heat generation, space on the chip, and signal delay. For this reason, Steiner minimal trees play an important role in VLSI design.

Due to manufacturing limitations, wires are typically restricted to having a finite set of fixed orientations. This has lead to an interest in the so called fixed orientation metrics, initially considered by Widmayer et al. [21], where the distance between a pair of points is the length of a shortest path between them using a fixed set of orientations.

Routing is typically performed in several layers with only one orientation allowed in each layer. Some layers may be more easily congested than others and thus, some orientations of wires may be less desirable than others [22]. For this reason, fixed orientatation metrics with a weight associated with each orientation have been considered.

Steiner minimal trees in the plane equipped with the rectilinear metric and more recently in general (weighted) fixed orientation metrics have received a great deal of attention [11, 13, 14, 7, 6, 18, 4, 5, 19]. A disadvantage of using Steiner minimal trees is that wire distance between some pairs of pins may be very large compared to the shortest possible distance. As a consequence, signal delay will be high between such pairs.

To obtain networks with small detours between any pair of points, spanners have been considered. For $t \geq 1$, a t-spanner for a set of points is a network interconnecting the points such that the distance in the network between any pair of the given points is at most t times longer than the shortest possible distance between them. The smallest t for which the network is a t-spanner is called the stretch factor of the network. Computing networks with small stretch factors is an active area of research. For more on spanners, see e.g. [17, 8, 20].

An interesting dual problem is the following: given a network interconnecting a set of n points in the plane, what is the stretch factor of this network?

Surprisingly, the fastest known algorithm for computing the stretch factor of a Euclidean network is a naive one that computes all-pairs shortest paths. If the network is planar, all-pairs shortest paths can be computed in $O(n^2)$ time [9], giving a quadratic time algorithm for computing the stretch factor of the network.

For simpler types of graphs, faster algorithms exist. For instance, it has been shown that the stretch factor of a path in the Euclidean plane can be found in $O(n \log n)$ *expected* time and that the stretch factor of trees and cycles can be found in $O(n \log^2 n)$ *expected* time [1, 15]. Using parametric search gives (rather complicated) $O(n\text{polylog } n)$ worst-case time algorithms for these types of networks.

To our knowledge, the problem of efficiently computing the stretch factor of networks in weighted fixed orientation metrics has not received any attention. Since these metrics may be used to approximate other metrics and due to their applications in VLSI design mentioned above, we believe this problem to be an important one.

In this paper, we give an $O(\sigma n \log^2 n)$ worst-case

*Department of Computer Science, University of Copenhagen, `koolooz@diku.dk`

time algorithm for computing the stretch factor of an n-vertex path embedded in the plane with a weighted fixed orientation metric defined by $\sigma \geq 2$ vectors. For the L_1-metric, we generalize the algorithm to $d \geq 3$ dimensions. Here, the running time is $O(n \log^d n)$. At the cost of an extra $\log n$-factor in running time, we show how to compute the stretch factor of trees and cycles. All algorithms have $O(n)$ space requirement. Compared to the complicated worst-case time algorithms for the Euclidean metric, our algorithms are relatively simple and should be easy to implement.

The organization of the paper is as follows. In Section 2, we make basic definitions and observations and introduce some notation. In Section 3, we consider the problem of computing the stretch factor of paths in the plane equipped with the L_1-metric. We give a new way of expressing the stretch factor which enables us to develop an efficient algorithm for this problem. Using simple linear transformations, we generalize the algorithm to arbitrary weighted fixed orientation metrics in Section 4 and in Section 5, we generalize it to higher dimensions. Using ideas of [15], we show how to efficiently compute the stretch factor of trees in Section 6. In Section 7, we present an algorithm that computes the stretch factor of cycles. In Section 8, we show how to modify the algorithms to compute a pair of vertices achieving the stretch factor of the path, tree, or cycle. Finally, we make some concluding remarks in Section 9.

2 Basic Definitions and Observations

Let G be a graph embedded in a metric space with metric d. For two vertices u and v in G, we define $d^G(u, v)$ as the d-length of a shortest path P between u and v in G, i.e.

$$d^G(u, v) = \sum_{(x,y) \in E_P} d(x, y),$$

where E_P is the set of edges of P. If u and v belong to distinct connected components of G then we define $d^G(u, v) = \infty$.

For distinct vertices u and v in G, the *detour* $\delta_G(u, v)$ between u and v in G is defined as $d^G(u, v)/d(u, v)$. The *stretch factor* δ_G of G is the maximum detour over all pairs of distinct vertices of G, i.e.

$$\delta_G = \max_{u,v \in G, u \neq v} \delta_G(u, v).$$

Given two subgraphs G_1 and G_2 of G, we define

$$\delta_G(G_1, G_2) = \max_{u \in G_1, v \in G_2, u \neq v} \delta_G(u, v).$$

Points and vectors in $\mathbb{R}^d$, $d \geq 2$, will be written in boldface. Unless otherwise stated, subsets of $\mathbb{R}^d$ that we consider are assumed to be closed.

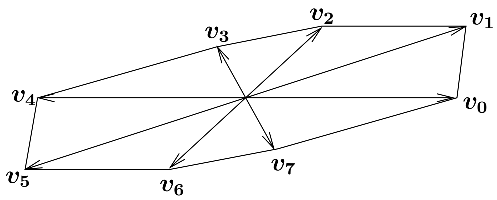

Figure 1: A unit circle in a weighted fixed orientation metric with $\sigma = 4$.

In all the following, let $\mathcal{V}$ denote a finite set of $\sigma \geq 2$ vectors $\boldsymbol{v_0}, \ldots, \boldsymbol{v_{\sigma-1}}$ all belonging to the upper half-plane. The *weighted fixed orientation metric* $d_{\mathcal{V}}$ is defined as follows. Letting $\boldsymbol{v_{\sigma+i}} = -\boldsymbol{v_i}$ for $i = 0, \ldots, \sigma-1$, the unit circle in $d_{\mathcal{V}}$ is the boundary of the convex hull of $\boldsymbol{v_0}, \ldots, \boldsymbol{v_{2\sigma-1}}$, see Figure 1.

We assume that all vectors, regarded as points, are on this boundary since any vectors that are not can be discarded without changing the metric.

Furthermore, we assume that vectors are ordered counter-clockwise starting with $\boldsymbol{v_0}$ and that $\boldsymbol{v}_0$, extends horizontally to the right. The latter can always be achieved by rotating the plane.

Drawing lines through the 2σ vectors partitions the plane into 2σ wedge-shaped regions. For $i = 0, \ldots, 2\sigma-1$, the region W_i defined by vectors $\boldsymbol{v_i}$ and $\boldsymbol{v_{(i+1) \bmod 2\sigma}}$ is called the *ith $\mathcal{V}$-cone*. For a point $\boldsymbol{p}$, we refer to $\boldsymbol{p}+W_i$ as the *ith $\mathcal{V}$-cone of $\boldsymbol{p}$*.

Let $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^2$. It can be shown that if $\boldsymbol{q}$ belongs to the ith $\mathcal{V}$-cone of $\boldsymbol{p}$ then a shortest path from $\boldsymbol{p}$ to $\boldsymbol{q}$ in the metric $d_{\mathcal{V}}$ consists of line segments each of which is parallel to either $\boldsymbol{v_i}$ or $\boldsymbol{v_{(i+1) \bmod 2\sigma}}$.

For $\boldsymbol{p} \in \mathbb{R}^2$, $r \geq 0$, we define $B_{\mathcal{V}}(\boldsymbol{p}, r)$ as the closed disc in $d_{\mathcal{V}}$ with center $\boldsymbol{p}$ and radius r, that is,

$$B_{\mathcal{V}}(\boldsymbol{p}, r) = \{\boldsymbol{q} \in \mathbb{R}^2 | d_{\mathcal{V}}(\boldsymbol{p}, \boldsymbol{q}) \leq r\}.$$

The L_1-metric in the plane is a special type of fixed orientation metric, defined by vectors $(1, 0)$ and $(0, 1)$. More generally, in $\mathbb{R}^d$, the L_1-distance between two points $\boldsymbol{p} = (p_1, \ldots, p_d)$ and $\boldsymbol{q} = (q_1, \ldots, q_d)$ is

$$L_1(\boldsymbol{p}, \boldsymbol{q}) = \sum_{c=1}^{d} |q_c - p_c|.$$

For $\boldsymbol{p} \in \mathbb{R}^d$ and $r \geq 0$, we let $B_1(\boldsymbol{p}, r)$ denote the closed L_1-ball in $\mathbb{R}^d$ with center $\boldsymbol{p}$ and radius r, i.e.

$$B_1(\boldsymbol{p}, r) = \{\boldsymbol{q} \in \mathbb{R}^d | L_1(\boldsymbol{p}, \boldsymbol{q}) \leq r\}.$$

3 Stretch Factor of Paths in the L_1-Plane

In this section, we show how to compute the stretch factor of an n-vertex path embedded in metric space $(\mathbb{R}^2, L_1)$ in $O(n \log^2 n)$ time and $O(n)$ space.

In the following, let $P = \boldsymbol{p_1} \rightarrow \boldsymbol{p_2} \rightarrow \cdots \rightarrow \boldsymbol{p_n}$ be an n-vertex path in the plane. We will make the simplifying assumption that all vertices of P are distinct. For if they were not, we would have $\delta_P = \infty$ and checking whether all vertices are distinct can be done in $O(n \log n)$ time.

In all the following, let $N = \{1, \ldots, n\}$. For $i \in N$ and $\delta > 0$, let $B_i(\delta)$ denote the L_1-disc $B_1(\boldsymbol{p_i}, r_i(\delta))$, where radius $r_i(\delta) = L_1^P(\boldsymbol{p_i}, \boldsymbol{p_n})/\delta$. The following lemma relates these discs to the stretch factor of P.

Lemma 1 *With the above definitions, the stretch factor of P is $\delta_P = \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall i, j \in N, i \neq j\}$.*

Proof. Let $\delta > 0$. For any $i, j \in N$,

$$\begin{aligned}\frac{L_1^P(\boldsymbol{p_i}, \boldsymbol{p_j})}{\delta} &= \frac{|L_1^P(\boldsymbol{p_i}, \boldsymbol{p_n}) - L_1^P(\boldsymbol{p_j}, \boldsymbol{p_n})|}{\delta} \\ &= |r_i(\delta) - r_j(\delta)|.\end{aligned}$$

Hence,

$$\begin{aligned}\delta_P < \delta &\Leftrightarrow \forall i, j \in N, i \neq j : L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) > |r_i(\delta) - r_j(\delta)| \\ &\Leftrightarrow \forall i, j \in N, i \neq j : L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) > r_i(\delta) - r_j(\delta) \\ &\Leftrightarrow \forall i, j \in N, i \neq j : L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) + r_j(\delta) > r_i(\delta) \\ &\Leftrightarrow \forall i, j \in N, i \neq j : B_j(\delta) \nsubseteq B_i(\delta).\end{aligned}$$

□

The idea of our algorithm is to see how much the size of the above defined L_1-discs can be increased before at least one of them includes another L_1-disc. By Lemma 1, this will then give us the stretch factor of path P.

For each $i \in N$ and for $w = 1, 2, 3, 4$, define $P_w(i)$ as the set of vertices of $P \setminus \{\boldsymbol{p_i}\}$ belonging to the wth quadrant of $\boldsymbol{p_i}$. Lemma 1 gives

$$\delta_P = \max_{w=1,2,3,4} \max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_w(i)\}.$$

Hence, δ_P is the maximum of four δ-values, one for each value of w. In the following, let us therefore restrict our attention to $w = 1$ and on computing

$$\max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_1(i)\} \qquad (1)$$

(the other quadrants are handled in a similar way).

The following lemma gives a useful way of determining whether $B_j(\delta)$ is contained in $B_i(\delta)$ when $\boldsymbol{p_j}$ belongs to the first quadrant of $\boldsymbol{p_i}$.

Lemma 2 *Let $\boldsymbol{p_i}$ be a given vertex and let $\boldsymbol{p_j} \in P_1(i)$. For $\delta > 0$, define $\boldsymbol{r_i}(\delta)$ and $\boldsymbol{r_j}(\delta)$ as the rightmost points in $B_i(\delta)$ and $B_j(\delta)$, respectively (see Figure 2). Then*

$$B_j(\delta) \subseteq B_i(\delta) \Leftrightarrow \boldsymbol{r_i}(\delta) \cdot (1, 1) \geq \boldsymbol{r_j}(\delta) \cdot (1, 1).$$

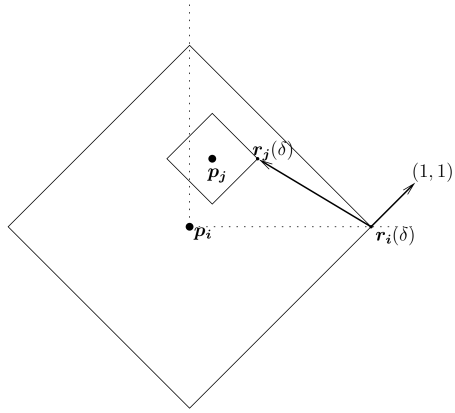

Figure 2: The situation in Lemma 2.

Proof. The point $\boldsymbol{r_j}(\delta)$ is to the right of $\boldsymbol{p_j}$ and belongs to the first quadrant of $\boldsymbol{p_i}$, implying that $L_1(\boldsymbol{p_i}, \boldsymbol{r_j}(\delta)) = L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) + r_j(\delta)$. Since $B_j(\delta) \subseteq B_i(\delta)$ if and only if $L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) + r_j(\delta) \leq r_i(\delta)$, we have

$$\begin{aligned}B_j(\delta) \subseteq B_i(\delta) &\Leftrightarrow L_1(\boldsymbol{p_i}, \boldsymbol{r_j}(\delta)) \leq r_i(\delta) \\ &\Leftrightarrow \boldsymbol{r_j}(\delta) \in B_i(\delta) \\ &\Leftrightarrow (\boldsymbol{r_j}(\delta) - \boldsymbol{r_i}(\delta)) \cdot (1, 1) \leq 0,\end{aligned}$$

since vector $(1, 1)$ is normal to the part of the boundary of $B_i(\delta)$ in the first quadrant of $\boldsymbol{p_i}$. □

Recall that, for any $i \in N$, $r_i(\delta) = L_1^P(\boldsymbol{p_i}, \boldsymbol{p_n})/\delta$. Hence, the dot product $\boldsymbol{r_i}(\delta) \cdot (1, 1)$ of Lemma 2 is an affine function of $1/\delta$, i.e. on the form $a(1/\delta) + b$, where a and b are constants. Denote this function by f_i.

Associate with each $\boldsymbol{p_i}$ a *lower envelope function* l_i of $1/\delta$, defined by

$$l_i(1/\delta) = \min\{f_j(1/\delta) | \boldsymbol{p_j} \in P_1(i)\}.$$

Recall that our goal is to compute (1). The following lemma relates this value to the intersection between f_i and l_i.

Lemma 3 *There is at most one intersection point between f_i and l_i on interval $]0, \infty[$. If $1/\delta'$ is such a point then*

$$\delta' = \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_1(i)\}.$$

If there is no intersection point then for any $\delta > 0$ and any $\boldsymbol{p_j} \in P_1(i)$, $B_j(\delta) \nsubseteq B_i(\delta)$.

Proof. Figure 3 illustrates the lemma.

For any point $\boldsymbol{p}$ in the first quadrant of $\boldsymbol{p_i}$, the value $\boldsymbol{p} \cdot (1, 1)$ is minimized when $\boldsymbol{p} = \boldsymbol{p_i}$. It follows that $f_i(1/\delta) < l_i(1/\delta)$ for all sufficiently small $1/\delta$. Hence, since the graph of l_i on $]0, \infty[$ is a chain of line segments (and one halfline) whose slopes decrease as we move

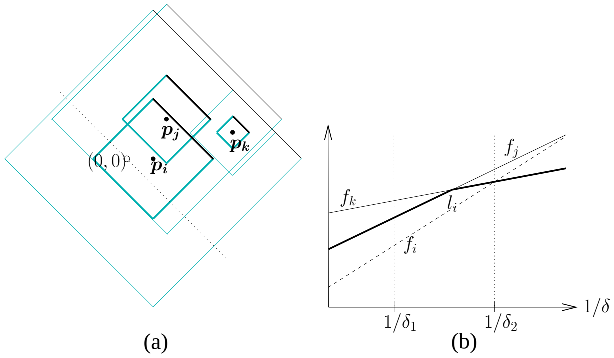

Figure 3: Illustration of Lemma 3. (a): L_1-discs $B_i(\delta)$, $B_j(\delta)$, and $B_k(\delta)$ for two values of δ: δ_1 (bold boundaries) and δ_2. (b): The corresponding functions f_i, f_j, and f_k. Lower envelope l_i is shown in bold. The distances in (a) between the dotted line and the black parts of L_1-discs correspond to values of functions f_i, f_j, and f_k at $1/\delta_1$ and $1/\delta_2$ in (b). Note that $B_k(\delta_2) \subseteq B_i(\delta_2)$. For all $1/\delta < 1/\delta_2$, $B_j(\delta) \nsubseteq B_i(\delta)$ and $B_k(\delta) \nsubseteq B_i(\delta)$.

from left to right, there is at most one intersection point $1/\delta'$ between f_i and l_i on interval $]0, \infty[$.

If intersection point $1/\delta'$ exists, the above shows that, on interval $]0, \infty[$, $f_i(1/\delta) < l_i(1/\delta')$ if $1/\delta < 1/\delta'$ and $f_i(1/\delta) > l_i(1/\delta')$ if $1/\delta > 1/\delta'$. And if no intersection point exists then f_i is below l_i on interval $]0, \infty[$.

Lemma 2 shows that $B_j(\delta) \subseteq B_i(\delta)$ if and only if $f_i(1/\delta) \geq f_j(1/\delta)$. Hence, $B_j(\delta) \nsubseteq B_i(\delta)$ for all $\boldsymbol{p_j}$ in the first quadrant $P_1(i)$ of $\boldsymbol{p_i}$ if and only if $f_i(1/\delta) < l_i(1/\delta)$. This shows the lemma. □

For each $i \in N$, let $\delta_i = 1/x_i$, where x_i is the intersection point between f_i and l_i. If no such point exists, set $\delta_i = 0$. Lemma 3 shows that (1) equals $\max_{i \in N} \delta_i$.

What remains therefore is the problem of computing the intersection (if any) between f_i and l_i for all i.

A naive algorithm for this problem computes, for each $i \in N$, lower envelope l_i in $O(n \log n)$ time (this is possible by Lemma 4 of Section 3.2) and then the intersection between f_i and l_i in $O(\log n)$ time. The total running time is $O(n^2 \log n)$.

A slightly faster algorithm computes, for each i, the intersection between f_i and f_j for each $\boldsymbol{p_j} \in P_1(i)$. The leftmost of these is then the intersection between f_i and l_i. This gives a total running time of $O(n^2)$.

Note that, for any $j \in N$, the lower envelope of f_j is f_j itself. Hence, the two algorithms above apply two extremes of the following strategy: for each $i \in N$, compute the leftmost of the intersections between f_i and lower envelopes associated with subsets of points in $P_1(i)$. The first algorithm considers, for each $i \in N$, only one subset (namely $P_1(i)$) whereas the second algorithm considers $|P_1(i)|$ subsets (each containing one element of $P_1(i)$).

In the next section, we present a faster algorithm which applies a strategy somewhere in between these two extremes.

3.1 The Algorithm

To simplify the description of the algorithm, we will leave out some of the details and return to them in Section 3.2 and Section 3.3, where we show how to obtain $O(n \log^2 n)$ running time and $O(n)$ space requirement.

The algorithm stores vertices of P in a balanced binary search tree $\mathcal{T}$ of height $\Theta(\log n)$ which is similar to a 1-dimensional range tree. Let V be the set of vertices of P. If V contains exactly one vertex, the root r of $\mathcal{T}$ is a leaf containing this vertex. Otherwise, r contains the median m of x-coordinates of vertices of V (in case of ties, order the vertices on the y-axis) and the subtree rooted at the left resp. right child of r is defined recursively for the set of vertices of V with x-coordinates less or equal to resp. greater than m.

Each node v of $\mathcal{T}$ corresponds to a subset S_v of vertices of P, namely those vertices stored at the leaves of the subtree of $\mathcal{T}$ rooted at v. We refer to these S_v-subsets as *canonical subsets*.

Note that each vertex $\boldsymbol{p_i}$ of P belongs to $\Theta(\log n)$ canonical subsets, namely those corresponding to vertices visited on the path from the root of $\mathcal{T}$ to the leaf containing $\boldsymbol{p_i}$.

In addition to a median, we associate with each node of $\mathcal{T}$ a lower envelope of line segments. This lower envelope is initially empty and will be dynamically updated during the course of the algorithm.

After having constructed $\mathcal{T}$, the algorithm makes a pass over the vertices of P in order of descending y-coordinate. In case of ties, vertices are visited from right to left.

The following invariant will be maintained throughout the course of the algorithm: *for each vertex v of $\mathcal{T}$, the lower envelope associated with v is the lower envelope of f_i-functions of vertices in S_v visited so far.*

When a vertex $\boldsymbol{p_i}$ of P is visited, the invariant is maintained by adding f_i to the $\Theta(\log n)$ lower envelopes associated with vertices on the path from the root of $\mathcal{T}$ to the leaf containing $\boldsymbol{p_i}$.

When the algorithm visits a vertex $\boldsymbol{p_i}$, it needs to find the intersection between f_i and l_i. As we saw earlier, explicitly computing l_i is too time-consuming.

Instead, we make use of our invariant which ensures that lower envelopes of visited vertices of all canonical subsets are given. The vertices in the first quadrant of $\boldsymbol{p_i}$ have all been visited and the set $P_1(i)$ of these vertices is therefore the union of visited vertices of the canonical subsets to the right of $\boldsymbol{p_i}$. So the intersection between f_i and l_i is the leftmost of the intersections between f_i and the lower envelopes associated with these canonical subsets, see Figure 4.

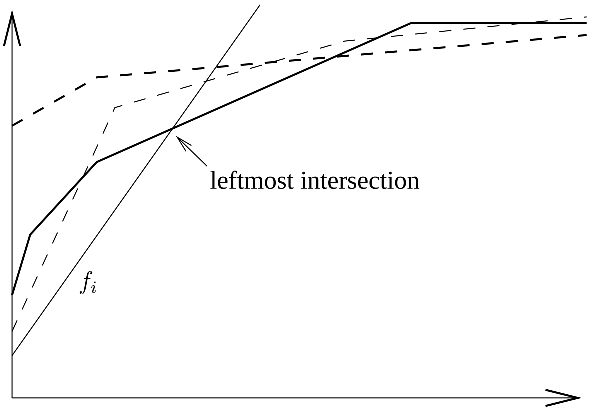

Figure 4: The intersection between f_i and l_i is the leftmost of the intersections between f_i and lower envelopes associated with canonical subsets to the right of $\boldsymbol{p_i}$.

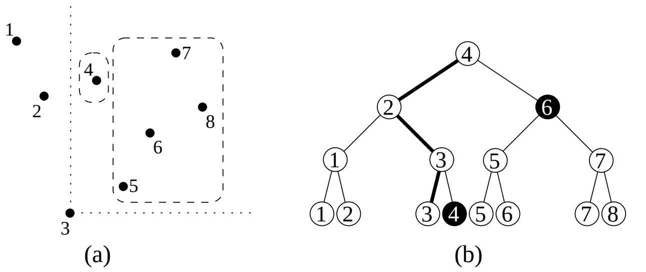

Figure 5: (a) Eight points shown with x-coordinates $1, \dots, 8$. The set of points with x-coordinate greater than 3 is the union of two canonical subsets. (b) The two canonical subsets are found by picking right children on the path (shown in bold) from the root of the range tree to the leaf with median 3. Picked children are coloured black.

Since canonical subsets may overlap, not all canonical subsets to the right of $\boldsymbol{p_i}$ are needed. The idea is to pick a small number in order to minimize running time.

The algorithm picks canonical subsets (or more precisely, nodes of $\mathcal{T}$ corresponding to canonical subsets) as follows. Let v_i be the leaf of $\mathcal{T}$ associated with $\boldsymbol{p_i}$. For each vertex v on the path from the root r of $\mathcal{T}$ to v_i, the canonical subset associated with the right child of v is picked unless this child itself is on the path from r to v_i, see Figure 5.

It is easy to see that the visited vertices in the union of the picked canonical subsets are exactly the vertices of $P_1(i)$. Since the height of $\mathcal{T}$ is $\Theta(\log n)$, the number of picked canonical subsets is $O(\log n)$.

One fine point: if there are vertices of P above $\boldsymbol{p_i}$ and with the same x-coordinate as $\boldsymbol{p_i}$, they may not all belong to the picked canonical subsets even though they belong to $P_1(i)$. We may ignore these however, since they will be picked when second quadrants are handled.

Intersections between f_i and each of the lower envelopes of the picked canonical subsets are then computed and the leftmost of these is picked as the intersection between f_i and l_i.

From these intersections, the value (1) is obtained. This is repeated for the other three quadrants, giving the stretch factor of P.

3.2 Running Time

In the description of the algorithm above, we left out some details. We now focus on them in order to analyze the running time of the algorithm.

It is easy to see that tree $\mathcal{T}$ can be constructed top-down in $O(n \log n)$ time. In the y-descending pass over the vertices of P, maintaining our invariant requires adding each f_i-function to $O(\log n)$ lower envelopes. Finding these lower envelopes takes $O(\log n)$ time by a traversal from the root to a leaf of $\mathcal{T}$ using the medians at vertices to guide the search. The following lemma shows that each insertion of a f_i-function into a lower envelopes takes $O(\log n)$ amortized time.

Lemma 4 *Let $l_1, \dots, l_k$ be k lines in the plane with positive slope and let L be the lower envelope of these lines on interval $]0, \infty[$. Constructing L incrementally can be done in $O(\log k)$ amortized time per line. Furthermore, L consists of at most $k-1$ line segments and exactly one halfline.*

Proof. We may assume that lines are added to L in the order $l_1, \dots, l_k$. For $i = 1, \dots, k$, let L_i be the lower envelope of $l_1, \dots, l_i$.

For $i > 1$, suppose that L_{i-1} has been computed and that the set Q_{i-1} of points on the graph of L_{i-1} where line segments meet are ordered from left to right in a red-black tree. Then computing the at most two intersections between l_i and L_{i-1} can be done in $O(\log i)$ time using two binary searches in the tree. When intersections have been found (if any), L_i is obtained from L_{i-1} in $O(q_{i-1} \log q_{i-1})$ time, where q_{i-1} is the number of points of Q_{i-1} that need to be removed to obtain L_i, i.e. the number of points of Q_{i-1} above l_i.

Since a point is removed at most once and each new line increases the number of points defining the lower envelope by at most two, it follows that the total time spent on constructing the lower envelopes in the order $L_1, \dots, L_k$ is $O(k \log k)$.

Since slopes of the line segments (and the halfline) defining the graph of L decrease from left to right, each of the k lines contribute with at most one line segment to the graph of L. Exactly one of the lines contribute with a halfline to the graph of L. Thus, the graph of L consists of at most $k-1$ line segments and exactly one halfline. □

Next, we need to analyze the time it takes to compute the intersection between f_i and l_i for each i. This involves picking $O(\log n)$ canonical subsets and computing the intersection between f_i and the lower envelopes associated with these subsets.

Clearly, the time it takes to find the canonical subsets is bounded by the height of $\mathcal{T}$ which is $\Theta(\log n)$. Since each lower envelope l is a monotonically increasing function and its graph consists of line segments (and one halfline), computing the intersection between f_i and l can be done in $O(\log n)$ time by using a data structure like a red-black tree to represent the ordered list of points of l where line segments meet.

It follows that our algorithm has $O(n \log^2 n)$ running time.

3.3 Improving Space Requirement

By Lemma 4, space requirement of our algorithm is $\Theta(n \log n)$ since this is the amount of space required to store all lower envelopes. We now show how to improve space requirement to linear without affecting running time.

We modify the algorithm so that, instead of making only one y-descending pass over the vertices of P, it makes $h(\mathcal{T})$ passes (for each of the four quadrants), where $h(\mathcal{T})$ is the height of $\mathcal{T}$.

In the kth pass, only lower envelopes at level k-nodes of $\mathcal{T}$ are updated; all other nodes of $\mathcal{T}$ contain empty lower envelopes. And only intersections between f_i-functions and lower envelopes at level k of $\mathcal{T}$ are computed.

The modified algorithm is correct since it computes exactly the same intersections as the old algorithm.

In each y-descending pass, the time spent on a vertex $\boldsymbol{p_i}$ of P is bounded by the time to add f_i to a lower envelope, and the time to compute the intersection between f_i and a lower envelope (if any). Hence, the total time spent in each pass is bounded by $O(n \log n)$. Since there are $h(\mathcal{T}) = \Theta(\log n)$ passes, the total running time is $O(n \log^2 n)$.

The modified algorithm has $O(n)$ space requirement since $\mathcal{T}$ has $O(n)$ nodes and since storing lower envelopes at one level of $\mathcal{T}$ requires $O(n)$ space by Lemma 4. This gives the first main result of the paper.

Theorem 5 *The stretch factor of an n-vertex path in $(\mathbb{R}^2, L_1)$ can be computed in $O(n \log^2 n)$ time and $O(n)$ space.*

4 Weighted Fixed Orientation Metrics

Recall that $d_{\mathcal{V}}$ denotes a weighted fixed orientation metric defined by a set $\mathcal{V}$ of $\sigma \geq 2$ vectors in the plane.

In this section, we generalize the algorithm of the preceding section to $d_{\mathcal{V}}$. The idea is simple: we apply a certain linear transformation to the vertices of P so that ith $\mathcal{V}$-cones are mapped to first quadrants and then use the algorithm of Section 3. This is done for all $\mathcal{V}$-cones, giving an algorithm with $O(\sigma n \log^2 n)$ time and $O(n)$ space requirement.

First, we observe that Lemma 1 also applies to the weighted fixed orientation metrics: simply define $B_i(\delta)$ as $B_{\mathcal{V}}(\boldsymbol{p_i}, r_i(\delta))$, where $r_i(\delta) = d^P_{\mathcal{V}}(\boldsymbol{p_i}, \boldsymbol{p_n})/\delta$, and replace L_1 by $d_{\mathcal{V}}$ in the proof. This gives us

$$\delta_P = \max_{1 \leq w \leq 2\sigma} \max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_w(i)\},$$

where $P_w(i)$ is the set of vertices of $P \setminus \{\boldsymbol{p_i}\}$ belonging to the wth $\mathcal{V}$-cone of $\boldsymbol{p_i}$. We restrict our attention to computing

$$\max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_1(i)\} \qquad (2)$$

(the other $\mathcal{V}$-cones are handled in a similar way).

Let $\boldsymbol{v_0}$ and $\boldsymbol{v_1}$ be the first and second vector of $\mathcal{V}$ respectively. It is easy to find the linear transformation T of the plane that maps $\boldsymbol{v_0}$ to $(1, 0)$ and $\boldsymbol{v_1}$ to $(0, 1)$. This allows us to generalize Lemma 2.

Lemma 6 *Let $\boldsymbol{p_i}$ be a given vertex and let $\boldsymbol{p_j} \in P_1(i)$. For $\delta > 0$, let $\boldsymbol{r_i}(\delta)$ and $\boldsymbol{r_j}(\delta)$ be the rightmost point in $B_i(\delta)$ and $B_j(\delta)$ respectively. Then*

$$B_j(\delta) \subseteq B_i(\delta) \Leftrightarrow T(\boldsymbol{r_i}(\delta)) \cdot (1, 1) \geq T(\boldsymbol{r_j}(\delta)) \cdot (1, 1),$$

where T is defined as above.

Proof. Let $\boldsymbol{v_0}$ and $\boldsymbol{v_1}$ be defined as above. Applying the ideas of the proof of Lemma 2, it follows easily that $B_j(\delta) \subseteq B_i(\delta)$ if and only if path $\boldsymbol{r_j}(\delta) \rightarrow \boldsymbol{r_i}(\delta) \rightarrow r_i(\delta)\boldsymbol{v_1}$ does not make a right turn at $\boldsymbol{r_i}(\delta)$.

Since T maps the triangle with corners $(0, 0)$, $\boldsymbol{v_0}$, and $\boldsymbol{v_1}$ to the triangle with corners $(0, 0)$, $(1, 0)$, and $(0, 1)$, linearity of T implies that $B_j(\delta) \subseteq B_i(\delta)$ if and only if path $T(\boldsymbol{r_j}(\delta)) \rightarrow T(\boldsymbol{r_i}(\delta)) \rightarrow T(r_i(\delta)\boldsymbol{v_1})$ does not make a right turn at $T(\boldsymbol{r_i}(\delta))$.

Since the vector from $T(\boldsymbol{r_i}(\delta))$ to $T(r_i(\delta)\boldsymbol{v_1})$ and vector $(-1, 1)$ have the same orientation,

$$\begin{aligned} B_j(\delta) \subseteq B_i(\delta) &\Leftrightarrow (T(\boldsymbol{r_j}(\delta)) - T(\boldsymbol{r_i}(\delta))) \cdot \widehat{(-1, 1)} \geq 0 \\ &\Leftrightarrow (T(\boldsymbol{r_j}(\delta)) - T(\boldsymbol{r_i}(\delta))) \cdot (1, 1) \leq 0. \end{aligned}$$

□

By defining affine functions f_i by $f_i(1/\delta) = T(\boldsymbol{r_i}(\delta)) \cdot (1, 1)$ and l_i by

$$l_i(1/\delta) = \min\{f_j(1/\delta) | \boldsymbol{p_j} \in P_1(i)\}$$

for $i \in N$, Lemma 6 and the results of Section 3 show that the value (2) may be computed in $O(n \log^2 n)$ time using $O(n)$ space. Since there are 2σ $\mathcal{V}$-cones to consider, we thus obtain the following generalization of Theorem 5.

Theorem 7 *Let $\mathcal{V}$ be a set of $\sigma \geq 2$ vectors defining a weighted fixed orientation metric $d_{\mathcal{V}}$ on $\mathbb{R}^2$. Then the stretch factor of an n-vertex path in $(\mathbb{R}^2, d_{\mathcal{V}})$ can be computed in $O(\sigma n \log^2 n)$ time and $O(n)$ space.*

5 Higher Dimensions

In this section, we generalize the algorithm of Section 3 to higher dimensions. In metric space $(\mathbb{R}^d, L_1)$, $d \geq 3$, we will show how to compute the stretch factor of an n-vertex path in $O(n \log^d n)$ time using $O(n)$ space.

In the following, assume that path $P = \boldsymbol{p_1} \to \boldsymbol{p_2} \to \cdots \to \boldsymbol{p_n}$ is embedded in $(\mathbb{R}^d, L_1)$, where $d \geq 3$. For $i \in N$, let $B_i(\delta)$ and $r_i(\delta)$ be defined as in Section 3.

First, we observe that Lemma 1 also holds in d dimensions. This gives us

$$\delta_P = \max_{1 \leq w \leq 2^d} \max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_w(i)\},$$

where $P_w(i)$ is the set of vertices of $P \backslash \{\boldsymbol{p_i}\}$ belonging to the wth orthant[1] of $\boldsymbol{p_i}$ for some ordering of the orthants. We assume that

$$O_1(i) = \{\boldsymbol{p} \in \mathbb{R}^d | \boldsymbol{p}[c] \geq \boldsymbol{p_i}[c] \forall c\}$$

is the first orthant of $\boldsymbol{p_i}$ and we restrict our attention to computing

$$\max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_1(i)\} \qquad (3)$$

(the other orthants are handled in a similar way).

The following lemma generalizes Lemma 2 to higher dimensions.

Lemma 8 *Let $\boldsymbol{p_i}$ be a given vertex and let $\boldsymbol{p_j} \in P_1(i)$. For $\delta > 0$, let*

$$\boldsymbol{r_i}(\delta) = (\boldsymbol{p_i}[1] + r_i(\delta), \boldsymbol{p_i}[2], \boldsymbol{p_i}[3], \ldots, \boldsymbol{p_i}[d])$$
$$\boldsymbol{r_j}(\delta) = (\boldsymbol{p_j}[1] + r_j(\delta), \boldsymbol{p_j}[2], \boldsymbol{p_j}[3], \ldots, \boldsymbol{p_j}[d])$$

and let $\boldsymbol{e}$ be the d-dimensional vector with d ones. Then

$$B_j(\delta) \subseteq B_i(\delta) \Leftrightarrow \boldsymbol{r_i}(\delta) \cdot \boldsymbol{e} \geq \boldsymbol{r_j}(\delta) \cdot \boldsymbol{e}.$$

Proof. Similar to the proof of Lemma 2, we have

$$B_j(\delta) \subseteq B_i(\delta) \Leftrightarrow L_1(\boldsymbol{p_i}, \boldsymbol{r_j}(\delta)) \leq r_i(\delta)$$
$$\Leftrightarrow \boldsymbol{r_j}(\delta) \in B_i(\delta).$$

Let $B_i^1(\delta)$ be the part of the boundary of $B_i(\delta)$ belonging to $O_1(i)$. Then $B_i^1(\delta)$ contains the points $\boldsymbol{p_i} + r_i(\delta)\boldsymbol{e_j}$, $j = 1, \ldots, d$, where $\boldsymbol{e_j}$ is the jth unit vector. Since $\boldsymbol{r_i}(\delta) \in B_i(\delta)$, the hyperplane H containing $B_i^1(\delta)$ is defined by $H = \{\boldsymbol{p} \in \mathbb{R}^d | (\boldsymbol{p} - \boldsymbol{r_i}(\delta)) \cdot \boldsymbol{e} = 0\}$.

Since $\boldsymbol{p_i}$ is an interior point of $B_i(\delta)$ and since $(\boldsymbol{p_i} - \boldsymbol{r_i}(\delta)) \cdot \boldsymbol{e} < 0$, it follows that, for any point $\boldsymbol{p} \in P_1(i)$, $(\boldsymbol{p} - \boldsymbol{r_i}(\delta)) \cdot \boldsymbol{e} \leq 0$ if and only if $\boldsymbol{p} \in B_i(\delta)$. Hence,

$$B_j(\delta) \subseteq B_i(\delta) \Leftrightarrow \boldsymbol{r_j}(\delta) \in B_i(\delta)$$
$$\Leftrightarrow (\boldsymbol{r_j}(\delta) - \boldsymbol{r_i}(\delta)) \cdot \boldsymbol{e} \leq 0.$$

□

For each $i \in N$, we define affine function f_i by $f_i(1/\delta) = \boldsymbol{r_i}(\delta) \cdot \boldsymbol{e}$, where $\boldsymbol{r_i}$ and $\boldsymbol{e}$ are defined as in Lemma 8 and we define lower envelope function l_i as in Section 3.

Since Lemma 3 holds with the f_i- and l_i-functions defined as above, it follows that the problem we are facing is to compute the intersection between f_i and l_i (if any) for all i. We deal with this problem in the next subsection where we generalize the algorithm of Section 3.1 to d dimensions.

5.1 The Algorithm

In Section 3.1, we used a 1-dimensional range tree. To compute the stretch factor of path P in d dimensions, we now consider a $(d-1)$-dimensional range tree.

First, a binary search tree $\mathcal{T}$ is constructed on the first coordinate of the vertices of P as described in Section 3.1. Each node v of $\mathcal{T}$ is associated with a $(d-2)$-dimensional range tree for the vertices in canonical subset S_v restricted to their last $d-1$ coordinates. This construction is repeated recursively for the $(d-2)$-dimensional range tree. The recursion stops when we reach a 1-dimensional range tree for coordinate $d-1$.

We associate lower envelopes only with nodes of the 1-dimensional range trees. Note that range trees are not defined for coordinate d. In this way, coordinates $d-1$ and d correspond to coordinates x and y in Section 3.1 respectively.

The algorithm then visits vertices of P in order of descending d-coordinate. In case of ties, vertices are visited from right to left on axis $d-1$.

Let $\boldsymbol{p_i}$ be the vertex currently being visited. The algorithm needs to update all lower envelopes corresponding to canonical subsets in 1-dimensional range trees that contain $\boldsymbol{p_i}$.

This is done as follows. All nodes of the $(d-1)$-dimensional range tree whose canonical subsets contain $\boldsymbol{p_i}$ are visited. Each of their associated $(d-2)$-dimensional range trees are visited recursively. When the nodes of a 1-dimensional range tree are visited, the function f_i corresponding to $\boldsymbol{p_i}$ is inserted into their associated lower envelopes.

For vertex $\boldsymbol{p_i}$, the algorithm also needs to compute intersections between f_i and lower envelopes corresponding to canonical subsets which are to the right of $\boldsymbol{p_i}$ on axes 1 to $d-1$.

This is done as follows. Let r be the root and let v_i be the leaf containing $\boldsymbol{p_i}$ in the $(d-1)$-dimensional

[1] An orthant is the higher dimensional equivalent of a quadrant. A d-dimensional coordinate system has 2^d orthants.

range tree. For each node v on the path from r to v_i the $(d-2)$-dimensional range tree associated with the right child of v is visited recursively unless this child itself is on the path from r to v_i. When reaching a 1-dimensional range tree, intersections between f_i and lower envelopes are found as described in Section 3.1.

The correctness of the above algorithm follows by generalizing the arguments of Section 3.1.

5.2 Running Time

We will now show that the algorithm described above has $O(n \log^d n)$ running time. In our analysis, we assume that dimension $d \geq 3$ is a constant.

As shown in [3], a $(d-1)$-dimensional range tree can be constructed in $O(n \log^{d-2} n)$ time. For each vertex $\boldsymbol{p_i}$ of P, finding the lower envelopes in which f_i is to be inserted takes $O(\log^{d-1} n)$ time. To see this, note that the algorithm recurses on $O(\log n)$ $(d-2)$-dimensional range trees associated with nodes of the $(d-1)$-dimensional range tree. For each of these range trees, the algorithm recurses on $O(\log n)$ $(d-3)$-dimensional range trees and so on. Thus, the total time to visit lower envelopes in which f_i is to be inserted is $O(\log^{d-1} n)$.

Since it takes $O(\log n)$ time to insert f_i into a lower envelope, the total time spent on inserting f_i-functions into lower envelopes is $O(n \log^d n)$.

A similar argument shows that it takes $O(n \log^d n)$ time to compute intersections between f_i-functions and lower envelopes. Hence, the total running time of the algorithm is $O(n \log^d n)$.

5.3 Improving Space Requirement

The above algorithm does not have linear space requirement. For instance, constructing the $(d-1)$-dimensional range tree using the algorithm of [3] requires $O(n \log^{d-2} n)$ space. By generalizing the idea of Section 3.3, we will modify our algorithm so that space requirement is improved to $O(n)$ without affecting running time.

The algorithm above makes one pass over the vertices of P in order of descending d-coordinate. We modify it so that it makes h_{d-1}^{d-1} passes, where h_{d-1} is the height of the $(d-1)$-dimensional range tree.

We enumerate the passes using a $(d-1)$-dimensional vector $\boldsymbol{C}$. Each entry of $\boldsymbol{C}$ is a number between 1 and h_{d-1}. Note that $h_{d-1} = \Theta(\log n)$ and that h_{d-1} is an upper bound on the height of all other range trees (since they all correspond to smaller sets of vertices of P).

Vector $\boldsymbol{C}$ can attain $\Theta(\log^{d-1} n)$ values and each of these values determine which parts of the range trees we are interested in in the current pass. More specifically, $\boldsymbol{C}[i] = k$ indicates that we are interested only in level k of all $(d-i)$-dimensional range trees, $i = 1, \ldots, d-1$. If some $(d-i)$-dimensional range tree has height less than $\boldsymbol{C}[i]$, it means that we are not interested in any levels of that tree in the current pass.

Consider a given pass of the vertices of P. We construct the $(d-1)$-dimensional range tree as before except that we only associate $(d-2)$-dimensional range trees with nodes at depth $\boldsymbol{C}[1]$. For each of these $(d-2)$-dimensional range trees, we only associate $(d-3)$-dimensional range trees with nodes at depth $\boldsymbol{C}[2]$ and so on. We refer to these range trees as *restricted range trees*.

Since canonical subsets associated with nodes at the same level of a range tree are disjoint, it follows easily that restricted range trees can be constructed in $O(n \log n)$ time (since d is assumed to be a constant). Thus, the total time spent on constructing restricted range trees over all passes is $O(n \log^d n)$.

In each pass, we only update those lower envelopes allowed by $\boldsymbol{C}$. The time spent on this for a given vertex $\boldsymbol{p_i}$ of P and a given pass is $O(\log n)$ since there is at most one lower envelope in which f_i is to be inserted in the current pass and it takes $O(\log n)$ time to find it and update it.

Similarly, computing the intersection between lower envelopes and a given vertex of P in a given pass takes $O(\log n)$ time. Thus, the total time spent in each pass is $O(n \log n)$. Since there are $\Theta(\log^{d-1} n)$ passes, the total running time of the modified algorithm is $O(n \log^d n)$, that is, the same as the original algorithm.

As for space requirement, we observe that the space required to represent restricted range trees is $O(n)$ and the space used for storing lower envelopes in a given pass is $O(n)$ since for a given vertex $\boldsymbol{p_i}$ of P, f_i is stored in at most one lower envelope in the current pass. This gives us the following generalization of Theorem 5.

Theorem 9 *The stretch factor of an n-vertex path in $(\mathbb{R}^d, L_1)$ can be computed in $O(n \log^d n)$ time and $O(n)$ space.*

6 Stretch Factor of Trees

In this section, we generalize our algorithms for paths to trees. We will show that the stretch factor of a tree with n vertices can be computed in $O(\sigma n \log^3 n)$ time for the weighted fixed orientation metrics and in $O(n \log^{d+1} n)$ time in the space $(\mathbb{R}^d, L_1)$ using $O(n)$ space.

Let us focus on space $(\mathbb{R}^d, L_1)$. The weighted fixed orientation metrics are handled in a similar way.

Let T be a tree with n vertices $\boldsymbol{p_1}, \ldots, \boldsymbol{p_n}$ embedded in $(\mathbb{R}^d, L_1)$. We use the idea of [15] to compute the stretch factor of T. First, T is partitioned into two subtrees T_1 and T_2 sharing a single vertex $\boldsymbol{p}$ such that each subtree contains between $n/4$ and $3n/4$ vertices. As shown in [15], this can be done in $O(n)$ time. Then the stretch factors of T_1 and T_2 are found recursively.

What remains is to determine value $\delta_T(T_1, T_2)$ if it is larger than either δ_{T_1} or δ_{T_2}.

Let I_1 and I_2 be indices of vertices in T_1 and T_2 respectively and let $I = I_1 \cup I_2$ be the indices of vertices in T. Let $m = \max_{i \in I_2} L_1(\boldsymbol{p_i}, \boldsymbol{p})$ and let $\delta > 0$. For each $i \in I$, let $B_i(\delta)$ be the L_1-ball $B_1(\boldsymbol{p_i}, r_i(\delta))$, where

$$r_i(\delta) = \begin{cases} (m + L_1^T(\boldsymbol{p_i}, \boldsymbol{p}))/\delta & \text{if } i \in I_1, \\ (m - L_1^T(\boldsymbol{p_i}, \boldsymbol{p}))/\delta & \text{if } i \in I_2. \end{cases}$$

Note that $r_i(\delta) \geq m$ for all $i \in I_1$ and $0 \leq r_i(\delta) \leq m$ for all $i \in I_2$.

Lemma 10 *With the above definitions,* $\delta_T(T_1, T_2) = \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall (i,j) \in I_1 \times I_2, i \neq j\}$. *Furthermore, for* $c = 1, 2$, $\delta_T \geq \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall i, j \in I_c, i \neq j\}$.

Proof. Let $\delta > 0$. For any $(i,j) \in I_1 \times I_2$, $i \neq j$,

$$\frac{L_1^T(\boldsymbol{p_i}, \boldsymbol{p_j})}{\delta} = \frac{L_1^T(\boldsymbol{p_i}, \boldsymbol{p}) + L_1^T(\boldsymbol{p_j}, \boldsymbol{p})}{\delta} = r_i(\delta) - r_j(\delta).$$

Hence,

$$\begin{aligned} &\delta_T(T_1, T_2) < \delta \Leftrightarrow \forall (i,j) \in I_1 \times I_2, i \neq j : \\ &\qquad L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) > \frac{L_1^T(\boldsymbol{p_i}, \boldsymbol{p_j})}{\delta} = r_i(\delta) - r_j(\delta) \\ &\qquad \Leftrightarrow \forall (i,j) \in I_1 \times I_2, i \neq j : B_j(\delta) \not\subseteq B_i(\delta). \end{aligned}$$

This shows the first part of the lemma.

To show the second part, let $\delta > 0$ and $i, j \in I_c$, $i \neq j$, be given. Since $r_i(\delta) - r_j(\delta) = (|L_1^T(\boldsymbol{p_i}, \boldsymbol{p}) - L_1^T(\boldsymbol{p_j}, \boldsymbol{p})|)/\delta$,

$$\begin{aligned} B_j(\delta) \subseteq B_i(\delta) \Rightarrow L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) &\leq \frac{|L_1^T(\boldsymbol{p_i}, \boldsymbol{p}) - L_1^T(\boldsymbol{p_j}, \boldsymbol{p})|}{\delta} \\ &\leq \frac{L_1^T(\boldsymbol{p_i}, \boldsymbol{p_j})}{\delta} \\ \Rightarrow \delta \leq \delta_T. \end{aligned}$$

It follows that

$$\begin{aligned} \delta_T &\geq \sup\{\delta > 0 | B_j(\delta) \subseteq B_i(\delta) \text{ for some } i, j \in I_c, i \neq j\} \\ &= \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall i, j \in I_c, i \neq j\}. \end{aligned}$$

□

Corollary 11 *With the above definitions,* $\delta_T \geq \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall i, j \in I, i \neq j\}$ *with equality if* $\delta_T = \delta_T(T_1, T_2)$.

Proof. Lemma 10 implies that

$$\begin{aligned} \delta_T &\geq \delta_T(T_1, T_2) \\ &= \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall (i,j) \in I_1 \times I_2, i \neq j\} \end{aligned}$$

and that, for $c = 1, 2$,

$$\delta_T \geq \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall i, j \in I_c, i \neq j\}.$$

□

Corollary 11 shows that if we pick the maximum of $\delta_T(T_1)$, $\delta_T(T_2)$, and the value

$$\inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall i, j \in I, i \neq j\}, \qquad (4)$$

then we obtain the stretch factor of T. Computing (4) is done exactly as for paths in $O(n \log^d n)$ time and $O(n)$ space.

It follows from the above that our algorithm has $O(\log n)$ recursion levels and uses $O(n \log^d n)$ time per level. This gives the following result.

Theorem 12 *The stretch factor of a tree with* n *vertices embedded in* $(\mathbb{R}^d, L_1)$ *can be computed in* $O(n \log^{d+1} n)$ *time and* $O(n)$ *space.*

The above arguments also apply to the weighted fixed orientation metrics, giving the following theorem.

Theorem 13 *Let* $\mathcal{V}$ *be a set of* $\sigma \geq 2$ *vectors defining a weighted fixed orientation metric* $d_{\mathcal{V}}$ *on* $\mathbb{R}^2$. *Then the stretch factor of a tree with* n *vertices in* $(\mathbb{R}^2, d_{\mathcal{V}})$ *can be computed in* $O(\sigma n \log^3 n)$ *time and* $O(n)$ *space.*

7 Stretch Factor of Cycles

We now show how to compute the stretch factor of an n-vertex cycle in $O(\sigma n \log^3 n)$ time for the weighted fixed orientation metrics and in $O(n \log^{d+1} n)$ time in the space $(\mathbb{R}^d, L_1)$ using $O(n)$ space.

We will restrict our attention to the space $(\mathbb{R}^2, L_1)$ since the weighted fixed orientation metrics are handled in a similar way and since generalizing the results of this section to higher dimensions follows from ideas similar to those of Section 5. So let C be a an n-vertex cycle $\boldsymbol{p_1} \to \boldsymbol{p_2} \to \cdots \to \boldsymbol{p_n} \to \boldsymbol{p_1}$ embedded in $(\mathbb{R}^2, L_1)$.

The problem of computing the stretch factor of C is harder than for paths since there are now two possible paths between each pair of distinct vertices.

To handle this, it will prove useful to replace C by a $2n$-vertex path $P = \boldsymbol{q_1} \to \cdots \to \boldsymbol{q_{2n}}$, where $\boldsymbol{q_i} = \boldsymbol{q_{n+i}} = \boldsymbol{p_i}$ for $i = 1, \ldots, n$.

For $i = 1, \ldots, 2n$ and for $\delta > 0$, let $B_i(\delta)$ denote the L_1-disc $B_1(\boldsymbol{q_i}, r_i(\delta))$, where $r_i(\delta) = L_1^P(\boldsymbol{q_i}, \boldsymbol{q_{2n}})/\delta$. Let m_C denote half the length of C.

With these definitions, we obtain the following result which is similar to Lemma 1.

Lemma 14 *With the above definitions, the stretch factor of* C *is* $\delta_C = \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall 1 \leq i, j \leq 2n, i \neq j, L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C\}$.

Proof. The proof follows by applying the ideas of the proof of Lemma 1 and from the observation that

$$\delta_C = \max\{\frac{L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j})}{L_1(\boldsymbol{q_i}, \boldsymbol{q_j})} | 1 \leq i, j \leq 2n, i \neq j, L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C\}.$$

□

For $1 \leq i \leq 2n$ and for $w = 1, 2, 3, 4$, define $P_w(i)$ as the set of vertices of $P \setminus \{\boldsymbol{p_i}\}$ belonging to the wth quadrant of $\boldsymbol{q_i}$. Lemma 14 gives

$$\delta_C = \max_{w=1,2,3,4} \max_{1 \leq i \leq 2n} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{q_j} \in P_w(i), L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C\}.$$

Hence, restricting our attention to first quadrants, the problem of computing

$$\max_{1 \leq i \leq 2n} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{q_j} \in P_1(i), L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C\} \quad (5)$$

is essentially the same as the problem of computing (1) of Section 3 except for one thing: only pairs of indices i, j, where $L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C$, are allowed.

Note that Lemma 2 also applies in this section. Let us therefore associate f_i- and l_i-functions to each vertex $\boldsymbol{q_i}$ of P. We define f_i as in Section 3 and define l_i by

$$l_i(1/\delta) = \min\{f_j(1/\delta) | \boldsymbol{q_j} \in P_1(i), L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C\}.$$

For $1 \leq i \leq 2n$, let $\delta_i = 1/x_i$, where x_i is the intersection point between f_i and l_i. If no such point exists, set $\delta_i = 0$. Then it follows easily from the results of Section 3 that (5) equals $\max_{1 \leq i \leq 2n} \delta_i$

What remains is to compute values x_i for all i. In the following, we describe an algorithm for this problem.

The algorithm stores vertices of P in a 1-dimensional range tree $\mathcal{T}$. Unlike in Section 3.1 however, vertices are not ordered in ascending x but by ascending distance to $\boldsymbol{q_1}$ in P. We will denote the leaves of $\mathcal{T}$ from left to right by $q_1, \ldots, q_{2n}$.

Let v be a node of $\mathcal{T}$ and let $\mathcal{T}_v$ be the subtree of $\mathcal{T}$ rooted at v. Associated with v is a 1-dimensional range tree for those vertices of P stored in the leaves of $\mathcal{T}_v$. This range tree is of the form described in Section 3.1 and will be updated in the same way.

Vertices of P are then considered in order of descending y (as in Section 3.1). Let $\boldsymbol{q_i}$ be the current vertex. Then range trees associated with nodes of $\mathcal{T}$ need to be updated w.r.t. $\boldsymbol{q_i}$. These nodes are the nodes on the path from the root of $\mathcal{T}$ to the leaf containing $\boldsymbol{q_i}$ since their associated range trees are exactly those that contain $\boldsymbol{q_i}$.

What remains in the processing of $\boldsymbol{q_i}$ is to compute the intersection between f_i and l_i. This involves computing intersections between f_i and lower envelopes in range trees associated with nodes of $\mathcal{T}$. However, only range trees containing vertices all of distance at most m_C to $\boldsymbol{q_i}$ in P should be considered.

Such range trees are picked as follows. First, $O(\log n)$ subtrees of $\mathcal{T}$ are picked such that they cover all leaves associated with vertices of P having distance at most m_C to $\boldsymbol{p_i}$. This is done using an algorithm similar to the range query algorithm of Section 5.1 of [3] with query range $[L_1^P(\boldsymbol{p_1}, \boldsymbol{p_i}) - m_C, L_1^P(\boldsymbol{p_1}, \boldsymbol{p_i}) + m_C]$. Then the range trees picked are those associated with roots of these subtrees.

The intersections between f_i and lower envelopes in the picked range trees are found as described in Section 3.1. The leftmost of these intersections is then the intersection x_i between f_i and l_i.

When all vertices of P have been considered in the y-descending path of vertices, the value (5) is found as $\max_{1 \leq i \leq 2n} \delta_i$, where $\delta_i = 1/x_i$.

The running time of the above algorithm is $O(n \log^3 n)$. This follows easily from the results of Section 3.2 and from the fact that the number of range trees considered for each vertex of P is $O(\log n)$.

Linear space requirement is obtained by making $O(\log^2 n)$ y-descending passes instead of one pass. In each pass, only range trees associated with nodes at a certain depth of $\mathcal{T}$ are considered and only nodes at a certain depth of each of the range trees associated with nodes of $\mathcal{T}$ are considered. We will leave out the details since they are similar to those of Section 3.3.

Generalizing the above to higher dimensions and to weighted fixed orientation metrics gives the following theorems.

Theorem 15 *The stretch factor of a cycle with n vertices embedded in $(\mathbb{R}^d, L_1)$ can be computed in $O(n \log^{d+1} n)$ time and $O(n)$ space.*

Theorem 16 *Let $\mathcal{V}$ be a set of $\sigma \geq 2$ vectors defining a weighted fixed orientation metric $d_{\mathcal{V}}$ on $\mathbb{R}^2$. Then the stretch factor of a cycle with n vertices in $(\mathbb{R}^2, d_{\mathcal{V}})$ can be computed in $O(\sigma n \log^3 n)$ time and $O(n)$ space.*

8 Finding a Vertex Pair Achieving the Stretch Factor

In the previous sections, we have considered the problem of computing the stretch factor of paths, trees, and cycles. Suppose that we are also interested in actually finding a pair of vertices for which the detour between them equals the stretch factor of the graph.

The algorithms described above are easily modified to find such a pair without affecting the time and space bounds. To achieve this, we make the following small change to the data structures defining lower envelope functions as follows. Let l be a lower envelope function considered during the course of the algorithm. Then we associate with every line segment (and halfline) of l the f_j-function defining this segment.

Now, suppose the stretch factor of the graph has been found. This value corresponds to a computed intersection between an f_i-function and a lower envelope function for some i. The above modification then allows us to find, in constant time, a vertex $\boldsymbol{p_j}$ such that the stretch factor of the graph is achieved as the detour between $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$.

In fact, we can obtain a slightly stronger result which we state in the following theorem.

Theorem 17 *Without affecting time and space bounds, all algorithms described above can be modified to compute, for every vertex $\boldsymbol{p_i}$, a vertex $\boldsymbol{p_j}$ maximizing the detour between $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$.*

9 Concluding Remarks

Given an n-vertex path P embedded in metric space $(\mathbb{R}^d, L_1)$, $d \geq 2$, we showed how to compute the stretch factor of P in $O(n \log^d n)$ worst-case time. For a general weighted fixed orientation metric in the plane, we gave an $O(\sigma n \log^2 n)$ time algorithm, where $\sigma \geq 2$ is the number of fixed orientations. We generalized our algorithms to trees and cycles at the cost of an extra $\log n$-factor in running time. All our algorithms have $O(n)$ space requirement.

An obvious question is whether our algorithms are optimal with respect to running time. In the Euclidean plane, an $\Omega(n \log n)$ lower bound is known for paths (and thus also for trees) and it is easily extended to the weighted fixed orientation metrics (for fixed σ). Thus, in the plane, there is a gap of $\log n$ for paths and $\log^2 n$ for trees between our time bounds and this lower bound.

It should be possible to modify the algorithms presented in [15] for computing the stretch factor of paths, trees, and cycles in the Euclidean plane to handle the rectilinear plane and possibly the more general weighted fixed orientation metrics. This would give an $O(n \log n)$ expected time algorithm for paths and an $O(n \log^2 n)$ expected time algorithm for trees and cycles. Is it possible to extend the ideas of this paper to handle other classes of graphs?

Finally, we believe that it is possible to handle more general fixed orientation metrics in higher dimensions using the ideas of this paper.

References

[1] P. K. Agarwal, R. Klein, C. Knauer, S. Langerman, P. Morin, M. Sharir, and M. Soss. Computing the Detour and Spanning Ratio of Paths, Trees and Cycles in 2D and 3D. *Discrete and Computational Geometry*, 39 (1): 17–37 (2008).

[2] S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. Smid. Euclidean spanners: short, thin, and lanky. *Proc. 27th ACM STOC*, 1995, pp. 489–498.

[3] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Computational Geometry - Algorithms and Applications (2nd ed.). *Springer-Verlag Berlin*, 1997, 2000.

[4] M. Brazil. Steiner Minimum Trees in Uniform Orientation Metrics. In *D.-Z. Du and X. Cheng, editor, Steiner Trees in Industries*, pages 1–27, Kluwer Academic Publishers, 2001.

[5] M. Brazil, D. A. Thomas, and J. F. Weng. Minimum Networks in Uniform Orientation Metrics. *SIAM Journal on Computing*, 30: 1579–1593, 2000.

[6] M. Brazil, P. Winter, and M. Zachariasen. Flexibility of Steiner Trees in Uniform Orientation Metrics. In *Proceedings of the 15th International Symposium on Algorithms and Computation (ISAAC), Lecture Notes in Computer Science 3341*, pp. 196–208, 2004.

[7] M. Brazil and M. Zachariasen. Steiner Trees for Fixed Orientation Metrics. *Technical Report 06-11, DIKU, Department of Computer Science, University of Copenhagen*, 2006.

[8] D. Eppstein. Spanning trees and spanners. In *J.-R. Sack and J. Urrutia, editors, Handbook of Computational Geometry*, pages 425–461, Elsevier Science Publishers, Amsterdam, 2000.

[9] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16 (1987), pp. 1004–1022.

[10] J. Gudmundsson, G. Narasimhan, and M. Smid. Fast pruning of geometric spanners. *STACS* 2005:508–520.

[11] M. Hanan. On Steiner's Problem with Rectilinear Distance. *SIAM Journal on Applied Mathematics*, 14(2):255–265, 1966.

[12] J. Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Information Processing Letters*, Vol. 33, no. 4, 1989, pp. 169–174.

[13] F. K. Hwang. On Steiner Minimal Trees with Rectilinear Distance. *SIAM Journal on Applied Mathematics*, 30:104–114, 1976.

[14] F. K. Hwang, D. S. Richards, and P. Winter. The Steiner Tree Problem. *Annals of Discrete Mathematics* 53, Elsevier Science Publishers, Netherlands, 1992

[15] S. Langerman, P. Morin, and M. Soss. Computing the maximum detour and spanning ratio of planar chains, trees and cycles. *Proceedings of the 19th International Symposium on Theoretical Aspects of Computer Science* (STACS '02), Lecture Notes in Computer Science, Vol. 2285, 2002, pp. 250–261.

[16] X. Y. Li and Y. Wang. Efficient construction of low weighted bounded degree planar spanner. *Int. J. Comput. Geometry Appl.* 14(1–2):69–84 (2004).

[17] G. Narasimhan and M. Smid. Geometric Spanner Networks. *Cambridge University Press*, 2007.

[18] B. K. Nielsen, P. Winter, and M. Zachariasen. An Exact Algorithm for the Uniformly-Oriented Steiner Tree Problem. In *Proceedings of the 10th European Symposium on Algorithms, Lecture Notes in Computer Science*, pp. 760–772, Springer, 2002.

[19] M. Sarrafzadeh and C. K. Wong. Hierarchical Steiner Tree Construction in Uniform Orientations. *IEEE Transactions on Computer-Aided Design*, 11: 1095–1103, 1992.

[20] M. Smid. Closest point problems in computational geometry. In *J.-R. Sack and J. Urrutia, editors, Handbook of Computational Geometry*, pages 877–935, Elsevier Science Publishers, Amsterdam, 2000.

[21] P. Widmayer, Y. F. Wu, and C. K. Wong. On Some Distance Problems in Fixed Orientations. *SIAM Journal on Computing*, 16(4): 728–746, 1987.

[22] M. C. Yildiz and P. H. Madden. Preferred Direction Steiner Trees. In *Proceedings of the 11th Great Lakes Symposium on VLSI (GLSVLSI)*, pages 56–61, 2001.