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

Christian Wulff-Nilsen*

Abstract

Let G be a graph embedded in the L_1-plane. The stretch factor of G is the maximum over all pairs of distinct vertices $\boldsymbol{p}$ and $\boldsymbol{q}$ of G of the ratio $L_1^G(\boldsymbol{p},\boldsymbol{q})/L_1(\boldsymbol{p},\boldsymbol{q})$, where $L_1^G(\boldsymbol{p},\boldsymbol{q})$ is the L_1-distance in G between $\boldsymbol{p}$ and $\boldsymbol{q}$. We show how to compute the stretch factor of an n-vertex path in $O(n\log^2 n)$ worst-case time and $O(n)$ space and we mention generalizations to trees and cycles, to general weighted fixed orientation metrics, and to higher dimensions.

1 Introduction

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. [3, 2, 4].

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

Fast algorithms for this problem are known only for simple graphs in the Euclidean plane. 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 a tree and a cycle can be found in $O(n\log^2 n)$ *expected* time [1]. Using parametric search gives (rather complicated) $O(n\text{polylog } n)$ worst-case time algorithms for these types of networks.

In the plane, a *weighted fixed orientation metric* [5, 6] is specified by a fixed set $\mathcal{V}$ of vectors and the distance between a pair of points is defined as the length of a shortest path between them consisting of line segments all with weighted orientations from $\mathcal{V}$.

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, we believe this problem to be an important one.

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

In this paper, we give an $O(n\log^2 n)$ worst-case time algorithm for computing the stretch factor of an n-vertex path embedded in the L_1-plane and we mention generalizations to trees and cycles, to arbitrary weighted fixed orientation metrics, and to higher dimensions. 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 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 and present an algorithm for this problem. In Section 4, we mention generalizations of our algorithm and we make some concluding remarks in Section 5.

2 Basic Definitions

For points $\boldsymbol{p}=(p_x,p_y)$ and $\boldsymbol{q}=(q_x,q_y)$, the L_1-*distance* $L_1(\boldsymbol{p},\boldsymbol{q})$ between $\boldsymbol{p}$ and $\boldsymbol{q}$ is $|q_x-p_x|+|q_y-p_y|$.

Let G be a connected graph embedded in the plane. To distinguish between a vertex of G and its location in the embedding, we write its location in boldface.

For two vertices u and v of G, define $L_1^G(\boldsymbol{u},\boldsymbol{v})$ as the length of a shortest path between u and v in G, where the length of a path is measured as the sum of L_1-lengths of the edges on the path in the embedding.

For distinct vertices u and v in G, the *detour* $\delta_G(\boldsymbol{u},\boldsymbol{v})$ between $\boldsymbol{u}$ and $\boldsymbol{v}$ in G is defined as $L_1^G(\boldsymbol{u},\boldsymbol{v})/L_1(\boldsymbol{u},\boldsymbol{v})$. The *stretch factor* δ_G of G is the maximum detour over all pairs of distinct vertices of G. If $\boldsymbol{u}=\boldsymbol{v}$ for two distinct vertices u and v of G, we define $\delta_G=\infty$.

For $\boldsymbol{p}\in\mathbb{R}^2$ and $r\geq 0$, we let $B_1(\boldsymbol{p},r)$ denote the closed disc in $(\mathbb{R}^2,L_1)$ with center $\boldsymbol{p}$ and radius r.

3 Stretch Factor of Paths

In the following, let $P=p_1\rightarrow p_2\rightarrow\cdots\rightarrow p_n$ be an n-vertex path embedded in $(\mathbb{R}^2,L_1)$. In this section, we show how to compute the stretch factor of P in $O(n\log^2 n)$ time and $O(n)$ space.

We will make the simplifying assumption that all vertices of P have distinct locations. For otherwise, we would have $\delta_P=\infty$ and checking whether all vertices have distinct locations 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 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 *The stretch factor of P is $\delta_P = \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta)$ for all $i, j \in N, i \neq j\}$.*

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

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

Hence,

$$\begin{aligned}\delta_P < \delta &\Leftrightarrow \forall i, j \in N, i \neq j : L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) > |r_i(\delta) - r_j(\delta)| \\ &\Leftrightarrow \forall i, j \in N, i \neq j : L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) > r_i(\delta) - r_j(\delta) \\ &\Leftrightarrow \forall i, j \in N, i \neq j : L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) + r_j(\delta) > r_i(\delta) \\ &\Leftrightarrow \forall i, j \in N, i \neq j : B_j(\delta) \not\subseteq B_i(\delta).\end{aligned}$$

□

The idea of our algorithm is to see how much the size of the above defined discs can be increased before at least one of them includes another 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

$$\delta_P = \max_{w=1,2,3,4, i \in N} \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_w(i)\}.$$

Hence, δ_P is the maximum of four δ-values, one for each value of w. By symmetry, we may restrict our attention to computing

$$\max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_1(i)\}. \quad (1)$$

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. Then*

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

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 $\boldsymbol{p_i}$. □

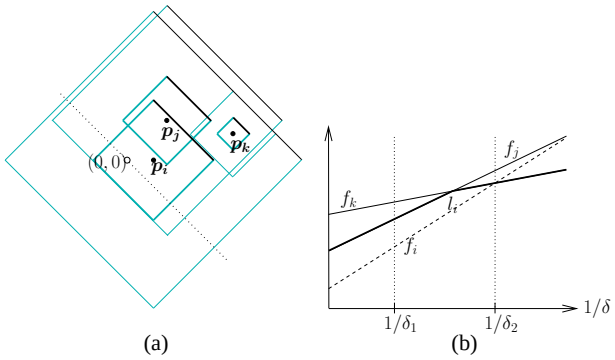

Figure 1: Illustration of Lemma 3. (a): L_1-discs $B_i(\delta)$, $B_j(\delta)$, and $B_k(\delta)$ for two values of δ: δ_1 (bold boundaries) and δ_2. (b): The corresponding functions f_i, f_j, and f_k. Lower envelope l_i is shown in bold. The distances in (a) between the dotted line and the black parts 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 $1/\delta < 1/\delta_2$, $B_j(\delta) \not\subseteq B_i(\delta)$ and $B_k(\delta) \not\subseteq B_i(\delta)$.

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

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

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*

$$\delta' = \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_1(i)\}.$$

If there is no intersection point then for any $\delta > 0$ and any $\boldsymbol{p_j} \in P_1(i)$, $B_j(\delta) \not\subseteq B_i(\delta)$.

Proof. Figure 1 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 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, $B_j(\delta) \not\subseteq B_i(\delta)$ for all $\boldsymbol{p_j}$ in the first quadrant $P_1(i)$ of $\boldsymbol{p_i}$ if and only $f_i(1/\delta) < l_i(1/\delta)$. 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. In the following, we describe an algorithm for this problem.

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, 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 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. However, 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.

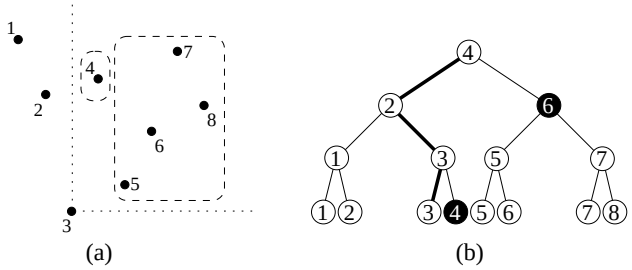

Figure 2: (a) Eight points shown with x-coordinates $1, \ldots, 8$. The set of points with x-coordinate greater than 3 is the union of two canonical subsets. (b) The two canonical subsets are found by picking right children (black) on the path from the root of the range tree to the leaf with median 3.

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 2.

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 $O(\log n)$.

A 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 and Space Requirement

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. It is easy to see that an f_i-function may be inserted in a lower envelope in $O(\log n)$ amortized time. Thus, the total time spent on maintaining the invariant is $O(n \log^2 n)$.

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

The algorithm described above has $O(n \log n)$ space requirement. To improve space requirement to linear, 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.

This gives the first main result of the paper.

Theorem 4 *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 Generalizations

In this section, we present generalizations of our algorithm. Proofs have been omitted due to space constraints.

By using ideas similar to those in [1], we obtain generalizations to trees and cycles. And by using higher dimensional range trees in Section 3.1, it is relatively easy to generalize our algorithm to $(\mathbb{R}^d, L_1)$. This gives the following result.

Theorem 5 *Let G be an n-vertex graph embedded in $(\mathbb{R}^d, L_1)$, $d \geq 2$. The stretch factor of G 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 cycle. Space requirement is $O(n)$.*

In two dimensions, we have generalized our algorithm to weighted fixed orientation metrics. Basically, we linearly map cones in such metrics to quadrants and then apply our algorithm for the L_1-metric.

Theorem 6 *Let G be an n-vertex graph embedded in the plane equipped with a weighted fixed orientation metric defined by $\lambda \geq 2$ weighted orientations. The stretch factor of G can be computed in $O(\lambda n \log^2 n)$ time when G is a path and in $O(\lambda n \log^3 n)$ time when G is a tree or cycle. Space requirement is $O(n)$.*

Suppose that we are not only interested in computing the stretch factor but also in finding a pair of vertices for which the detour between them equals the stretch factor of the graph. The following theorem shows that this can be done without affecting time and space bounds.

Theorem 7 *Within the same 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 $\boldsymbol{p_j}$ in the graph. In particular, a vertex pair achieving the stretch factor of the graph can be found within these time and space bounds.*

5 Concluding Remarks

Given an n-vertex path in $(\mathbb{R}^2, L_1)$, we showed how to compute its stretch factor in $O(n \log^2 n)$ worst-case time and $O(n)$ space. We mentioned generalizations to cycles and trees, to higher dimensions and to weighted fixed orientation metrics in the plane.

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 trees and it is easily seen to hold for the L_1-metric. Thus, there is a gap of $\log n$ for paths and $\log^2 n$ for trees between our time bounds and this lower bound. Is it possible to handle more general types of graphs?

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] 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.

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

[4] 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.

[5] K. J. Swanepoel. The Local Steiner Problem in Normed Planes. *Networks*, 36(2):104–113, 2000.

[6] 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.