Weighted Rectilinear Approximation of Points in the Plane

Mario A. Lopez[1] and Yan Mayster[2]

[1] University of Denver, Department of Mathematics, 2360 S. Gaylord St., Denver, CO 80208, USA
mlopez@du.edu

[2] University of Denver, Department of Computer Science, 2360 S. Gaylord St., Denver, CO 80208, USA
ymayster@cs.du.edu

Abstract. We consider the problem of weighted rectilinear approximation on the plane and offer both exact algorithms and heuristics with provable performance bounds. Let $S = \{(p_i, w_i)\}$ be a set of n points p_i in the plane, with associated distance-modifying weights $w_i > 0$. We present algorithms for finding the best fit to S among x-monotone rectilinear polylines $\mathcal{R}$ with a given number $k < n$ of horizontal segments. We measure the quality of the fit by the greatest weighted vertical distance, i.e., the approximation error is $\max_{1 \le i \le n} w_i d_v(p_i, \mathcal{R})$, where $d_v(p_i, \mathcal{R})$ is the vertical distance from p_i to $\mathcal{R}$. We can solve for arbitrary k optimally in $O(n^2)$ or approximately in $O(n \log^2 n)$ time. We also describe a randomized algorithm with an $O(n \log^2 n)$ expected running time for the unweighted case and describe how to modify it to handle the weighted case in $O(n \log^3 n)$ expected time. All algorithms require $O(n)$ space.

1 Introduction

The approximation of points in the plane using piecewise linear functions has drawn much interest from researchers in computational geometry and other fields. Many variants exist as a result of different constraints on the nature of the approximating curve, its complexity, error metric or the quality of the approximation. For a sample of recent results as well as references to other relevant work see [1, 4–6, 9, 12]. For these variants, two subclasses of problems can be considered. The first, min-#, calls for a solution curve with the least number of line segments (in the rectilinear case, only horizontal segments are counted) given a target error ε. The second, min-ε, specifies a number k and asks for a curve with no more than k segments that achieves least possible error ε. Finally, we can also add an additional restriction in the form of weights attached to individual points, which modify the distances from the points to the approximating line (usually, as multiplicative constants). This restriction creates many new versions of the problem (see [8, 16]). This paper addresses the weighted min-ε problem of approximating a set of points by a rectilinear curve using the min-max vertical distance metric.

In defining the problem we use much of the same notation as in [3], which was the first to tackle the unweighted case. Let $S = \{p_i = (x_i, y_i), i = 1, \ldots, n\}, x_1 < x_2 < \ldots < x_n$, be a set of n points in the plane. For $1 \leq i \leq j \leq n$, define $S_{ij} := \{p_i, p_{i+1}, \ldots, p_j\}$. A curve $\mathcal{R}$ is rectilinear if it consists only of alternating horizontal and vertical segments and is x-monotone if the x-domains of any two consecutive horizontal segments meet in a single value. From now on, when we speak of approximation curves they are both rectilinear and x-monotone.

We now define *vertical distance*, the error function used in our method. If a horizontal segment s has y-coordinate y_s and x-range $[x_s, x'_s]$, then the weighted vertical distance from s to a point p_i with associated weight w_i is

$$d_v^W(p_i, s) = \begin{cases} w_i|y_i - y_s| \text{ if } & x_i \in [x_s, x'_s], \\ \infty & \text{otherwise.} \end{cases}$$

Then, the weighted vertical distance between a point p_i and a curve $\mathcal{R}$ is defined as

$$d_v^W(p_i, \mathcal{R}) = \min_{s \in \mathcal{R}} d_v^W(p_i, s).$$

In the spirit of [3], the eccentricity of $\mathcal{R}$ with respect to S is the maximum vertical error between the points of S and $\mathcal{R}$, i.e.,

$$e(S, \mathcal{R}) = \max_{1 \leq i \leq n} d_v^W(p_i, \mathcal{R}).$$

A point p_i of S is said to be "covered" by a horizontal segment s of $\mathcal{R}$ if x_i is in the x-domain of s. We can see that every point of S is covered by some horizontal segment of $\mathcal{R}$ and the set of all points covered by s is some S_{ij} (which, as in [3], we call the *allocation set* of s). All allocation sets can be assumed nonempty as, otherwise, we unnecessarily increase the complexity of $\mathcal{R}$. Furthermore, the boundaries between adjacent horizontal segments can be fixed arbitrarily in the intervals between adjacent allocation sets.

Díaz-Báñez and Mesa [3] provide an $O(n^2 \log n)$ algorithm to solve the unweighted min-ε problem using their $O(n)$ solution for the min-# problem. They solve the min-# problem by sweeping the points from left to right and extending the current segment while the y-span of the points it covers is at most twice the allowed eccentricity. Thereafter, they solve min-ε by reducing it to a binary search on the "candidate" eccentricities (which number $O(n^2)$, one for each possible pair of points p_i, p_j, $i \leq j$). Later, Wang [15] reduces the time for min-ε to $O(n^2)$ by carefully generating a set of at most $2n - 2$ candidate errors which includes the optimal one and running the linear min-# algorithm on each of these errors.

Mayster and Lopez [10] improve over Wang with a min-ε algorithm that runs in $O(\min\{n^2, nk \log n\})$ time. Their algorithm uses Wang's $O(n)$ candidate eccentricities coupled with an auxiliary tree structure that cuts the time for each min-# instance down to $O(k \log n)$. The second result of [10] is a greedy heuristic (GCSA) that runs in $O(n \log n)$ time. It can generate curves with $2k-1$ segments with eccentricity no worse than that for an optimal curve consisting of

k segments as well as produce curves with k segments with eccentricity within a factor of 3 from k-optimal.

The rest of the paper is organized as follows. In the next section we introduce the dual perspective to modelling weighted distances from points to an approximating segment. In Section 3, we describe an exact algorithm that runs in $O(n^2)$ time. In Section 4 we discuss a modified GCSA heuristic that utilizes the dual perspective to maintain the costs efficiently. It runs in $O(n \log^2 n)$ time and has the same error bounds with proofs carrying over from [10]. Finally, in Section 5 we describe a randomized algorithm that solves the unweighted (resp. weighted) version of the problem in $O(n \log^2 n)$ (resp. $O(n \log^3 n)$) expected time.

2 Preliminaries

First, we consider the optimal placement of a horizontal segment with respect to its (fixed) allocation set. In the unweighted case the error is minimized when the segment is centered with respect to the y-range of the points. Thus, the optimal location of the horizontal segment is unique and can be determined from two points in the allocation set. This is still true in the weighted scenario, but the optimal location may not correspond to the midpoint of the y-range. However, it must still be equidistant (under weighted distance) from the furthest points above and below it, as otherwise a small shift in its position would decrease the error.

There cannot be two different locations for the optimal segment because of the semi-monotonicity of the distance function. If two distinct segments s and s' were both optimal, then the distance from s' to one of the two points that define s would be greater than the distance from s to that point, in violation of the optimality of s'. If the two points that define the y-coordinate y_s of the best approximating segment s have coordinates $(x_i, y_i), (x_j, y_j)$ and corresponding weights w_i, w_j, then y_s is given by

$$(y_i - y_s)w_i = (y_s - y_j)w_j \Rightarrow y_s = \frac{y_i w_i + y_j w_j}{w_i + w_j}.$$

Therefore, the solution to the problem is the intersection of two lines $c = -w_i y + y_i w_i$ and $c = w_j y - y_j w_j$, where c stands for the cost of approximating the point by a segment located at y. This leads us to consider a "cost-location" space composed of such lines, each point in S giving rise to one upward and one downward sloping line with the absolute values of the slopes equal to the weight of the point. Let us suppose that all points in S are located in the first quadrant, i.e. $x_i, y_i > 0 \; \forall 1 \leq i \leq n$. We map each point p_i with the corresponding weight w_i to the pair of lines in the "cost-location" plane $\ell_{i0} = w_i y_i - w_i y$ and $\ell_{i1} = -w_i y_i + w_i y$ and restrict their domain to the first quadrant. Thus, for each point we have a linear transformation ℓ_i of the absolute value metric function restricted to the nonnegative domain. Each such wedge shaped function ℓ_i computes the distance from p_i to the approximating segment as we hypothetically sweep it upward starting from $y = 0$ and consists of a finite down-sloping segment (recording the

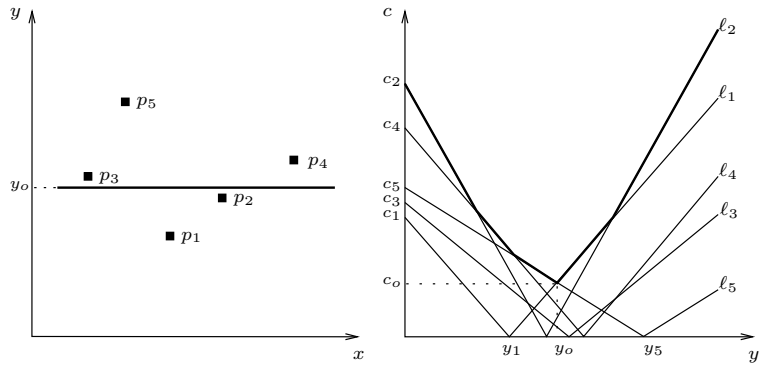

Fig. 1. (a) A set of points $p_i = (x_i, y_i)$, numbered by increasing y-coordinate, having respective weights w_i such that $w_2 > w_4 > w_1 > w_3 > w_2$, and the best fit segment. (b) The corresponding lines in the cost-location plane with the slopes w_i and the vertical axis intercepts c_i. The lowest point of the envelope is identified with the cost and y-coordinate of the best fit segment.

cost for $y < y_i$) and an infinite up-sloping ray (for the cost when $y > y_i$). Which portion of this arrangement of $2n$ cost lines keeps track of the greatest distance (i.e., the furthest point) to the approximating segment for any segment position y? The answer is quite obvious - the upper envelope of the arrangement is made up of the segments of the cost lines of those points that at some y are furthest from the approximating segment. The optimal location is given by the lowest point, which is also the lowest vertex, of the upper envelope.

We observe that in the case when S is known and fixed the above-mentioned problem of finding the lowest vertex of the upper envelope has been tackled successfully before, as it is nothing other than finding the optimal solution to a linear program in 2D. The best known deterministic algorithm for this has been developed by [11] and runs in $O(n)$ time. In addition, a very simple randomized algorithm [14] exists that has $O(n)$ expected running time. In our optimal algorithm we shall need to solve this problem repeatedly for each new subset of S, which differs in a single point from the previous subset, in order to compute the eccentricities of candidate curves and, therefore, using the $O(n)$ algorithm as a subroutine is an overkill.

However, there are more efficient algorithms to dynamically maintain common intersections of half-planes. In particular, a clever dynamization technique by Overmars and van Leeuwen [13] can be exploited to maintain the upper envelope in $O(\log^2 n)$ time per update (insertion or deletion of a line) and enables us to query for the lowest point on the boundary in just $O(\log n)$ time. The essence of their approach is to store the "left" half-planes (i.e., those that contain the left ray of any horizontal line) and the "right" half-planes in two separate augmented binary search trees. The lines bounding the half-planes are stored at the

leaves and ordered by slope. In our case, since each point contributes an entire wedge with both bounding lines having the same (in absolute value) slope, it makes sense to have just one tree and store the points themselves at the leaves sorted by weight. Then, the bounding lines of the left half-planes are sorted in descending order and the bounding lines of the right half-planes are sorted in ascending order (without explicitly storing these lines). As per [13], each internal node is augmented with a pointer to the parent and the largest slope value (largest point weight) of the lines in its left subtree (needed for concatenation). Most importantly, the portion of the upper envelope of the left half-plane lines in its subtree that does not contribute to the upper envelope of the left half-planes of its parent is stored in a concatenable queue along with the number of lines on its envelope that belong to the upper "left" envelope of the parent. The "right" upper envelope is handled similarly, so each internal node has two concatenated queues associated with it.

Then, the overall "left" upper envelope is stored at the root of the "left" tree (and, similarly, the "right" upper envelope is stored at the root of the "right" tree). Using the procedures DOWN and UP described in [13] one can insert and delete lines and maintain the queue structures as well as the balance of the tree. Then, the intersection of the left and right envelopes can be found efficiently in $O(\log n)$ time as is also proven in the original paper.

Finally, we note that there are other dynamic half-plane intersection algorithms that outperform the above-mentioned algorithm by Overmars and Leeuwen and run in $O(n \log n)$ amortized time, such as [7] and [2].

3 An Exact Algorithm

As observed in the previous section, the error of each approximating segment in its best position is determined by two points and, therefore, so is the eccentricity of the curve. It is still valid to use Wang's choice of candidate eccentricities and then it remains to describe how to compute these and the candidate curves that they give rise to. In Wang's algorithm, when one of the two pointers (called sweep lines in the original paper) is advanced, the error of the best approximating segment for the set of points between the two pointers is computed. This error computation in the weighted distance case corresponds to finding the lowest point on the upper envelope of the wedge lines in the cost-location plane as these lines are added or deleted one at a time. As mentioned in the previous section, computing the candidate eccentricities can be done using the $O(\log^2 n)$ dynamic half-plane intersection algorithm of [13].

We now turn to the question of how to compute a candidate curve itself once the target eccentricity ε has been found. This can be done with a slightly modified min-# algorithm of [3]. In this new version, each point (x_i, y_i) with the weight w_i is represented by a vertical line segment $v_i = (x_i, y_i - \frac{\varepsilon}{w_i})(x_i, y_i + \frac{\varepsilon}{w_i})$. Then, the algorithm proceeds in essentially the same way as described in [3]. We build horizontal segments of the curve by piercing consecutive vertical segments v_i. At first, we initialize the allocation set of the first horizontal segment to the single

point (x_1, y_1) and define its corridor to be $(y_{min} = y_1 - \frac{\varepsilon}{w_1}, y_{max} = y_1 + \frac{\varepsilon}{w_1})$. Then, adding each additional point p_i to the allocation set causes the segment's corridor to be updated to $y'_{min} = \max\{y_{min}, y_i - \frac{\varepsilon}{w_i}\}, y'_{max} = \min\{y_{max}, y_i + \frac{\varepsilon}{w_i}\}$. We keep extending the current horizontal segment of the curve for as long as adding new points does not cause the corridor to become empty, i.e. until further expansion of the allocation set would make $y'_{min} > y'_{max}$. Therefore, computing both the candidate eccentricity and curve takes $O(n)$ time leading to the following result.

Theorem 1. *The weighted rectilinear approximation problem can be solved in $O(n^2)$ time.*

This time bound becomes considerably reduced if the number of distance weights associated with the points of S is equal to a constant. In this case, the line wedges in the cost-location plane only have a constant number of distinct slopes. It is easy to see that for any given slope only one line wedge with that slope may contribute to the downward (and, similarly, upward) portion of the upper envelope. Furthermore, in our case, it is obvious that only the line wedge that contributes the first segment to the downward portion may also contribute a segment to the upward portion (due to the fact that all other line wedges that are part of the downward portion have smaller slope and a further x-intercept than the first one). All other line wedges may contribute only to one of the two portions. This means that the upper envelope consists of no more than $n + 1$ segments. In the case of a constant number of slopes c, we have no more than $c+1$ segments on the envelope and, therefore, the above algorithm runs in linear time. This is summarized in the next theorem.

Theorem 2. *The weighted rectilinear approximation problem with a constant number c of distance-modifying weights can be solved in $O(cn)$ time.*

4 A Heuristic with Provable Bounds

In [10], the authors describe a simple yet in practice quite accurate GCSA approximation algorithm for the problem of rectilinear curve fitting. The algorithm begins by building a curve consisting of n singleton segments and computes the costs that would result from merging the allocation sets of each adjacent pair of such segments. These costs are prioritized by storing them in a min-heap and, subsequently, at each iteration the minimum cost is extracted and the pair of associated segments is merged. The algorithm then updates the structure and the costs that involve the newly created enlarged segment and its neighbors.

We now modify this algorithm to be able to solve the weighted version of the same problem. While the overall structure of the algorithm shall remain unchanged, we have to supply new details for the merge step and analyze how these affect the overall running time. Now merging two allocation sets can no longer be accomplished in constant time as the points responsible for the error of the new larger segment are not necessarily a subset of the points defining the

placement of the old segments. Recall that the y-coordinate of the new longer segment s is determined by a pair of points whose so-called cost lines in the cost-location plane define the lowermost point of the upper envelope of all such cost lines coming from the points in the allocation set of s. Clearly, the cost lines that define this point come from the upper envelopes of the old segments' cost lines. Hence, the placement of the new longer segment can be determined by any two points whose cost lines were on the upper envelopes of their respective segments. We, therefore, have to keep track of the points defining these upper envelopes for each allocation set (upper envelope points).

Each of these points contributes at most two edges to the upper envelope and no two edges on the same envelope have overlapping x-ranges except at the boundaries. We can, therefore, store these in a binary tree ordered by x-range with pointers going to the original points. We also note that ordering the edges by x-range also has the effect of sorting them by slope as well as inducing a semi-sorted order on their y-ranges, since these decrease until the lowest point on the envelope and then monotonically increase. Furthermore, all upper envelopes are necessarily concave down, an important property that will be of use later.

When "merging" the allocation sets of two curve segments, their upper envelopes S (for *small*) and B (for *big*) need to be "merged" to produce the upper envelope of the new segment. Suppose that $|B| = n, |S| = m$ and $n > m$ (where the cardinality of an envelope is equal to the number of lines contributing segments to it). When we merge S and B, we always traverse S sequentially and B sometimes sequentially (when B is below S) and sometimes logarithmically (when B is above S). Clearly, the segments that survive (either partially or in their entirety) are on the upper envelope of $S \cup B$. Therefore, we need to find all points of intersection between S and B (for this is where they switch roles, one going below the other) and stitch together those portions that contribute to the overall upper envelope. Hence, when B is below S, we remove segments from B one by one (in $O(\log n)$ time per segment) and replace them by segments from S. Once removed, these lines (i.e., points in the allocation set) will never contribute to the upper envelope. When B is above S, that portion of B needs to be preserved and traversing it sequentially in order to find the next intersection between S and B would lead to a linear amortized time per merge and, thus, to the total quadratic time for the entire algorithm (consisting of $O(n)$ merges).

We begin with the leftmost segments of S and B. As we move along S, for each of its segments s with endpoints p_l, p_r we locate (via a binary search) the segments b_l, b_r (potentially, $b_l = b_r$) in B whose x-ranges contain the x-coordinates x_l, x_r of those endpoints. In the case of ties, when the endpoints of two segments of B have x-coordinate x_l or x_r, we always pick the segment of B that begins at x_l and ends at x_r. We then test if p_l is above or below b_l and, similarly, whether p_r is above or below b_r. If p_l or p_r coincide with the endpoints of b_l or b_r, we test whether s itself is below or above b_l or b_r. If both p_l and p_r (or s itself in the case of coinciding endpoints) are above the segments of B, then because of the concavity of upper envelopes we know that s is completely above B (Figure 2a) and, therefore, it must be added to the upper envelope of $S \cup B$

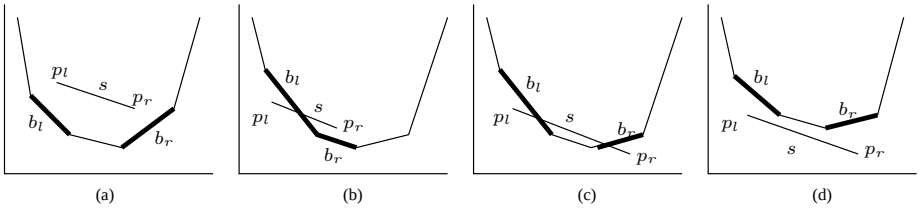

Fig. 2. (a) Case I: both endpoints of s are above the bigger envelope B. (b) Case II: p_l is below a segment b_l of B while p_r is above B (same as p_l above B and p_r below). (c) Case III: both endpoints of s are below B and an intersection exists. (d) Case IV: endpoints of s are as in Case III but there is no intersection.

and all segments of B from b_l to b_r (except, perhaps, b_r itself if its x-range is not completely covered by the x-range of s) can be removed from consideration. As a way to simplify and speed up the process, we create the upper envelope of $S \cup B$ completely inside of the data structure for B. Therefore, all deletions of segments of B and insertions of the segments of S are carried out straight on the binary tree containing B with the result that after the merge is complete B contains the final "merged" envelope.

If one of the endpoints of s (again, in the case of endpoints coinciding, s itself) is above B and the other is below B, then an intersection exists (Figure 2b) and can be found in logarithmic time by simply doing a binary search on the segments of B and testing them as being above or below s, or simply walking along B starting from the segment which is below s and deleting segments from B until we arrive at the intersection at which point we link up with s. Thus, all segments of B below s are removed (again, except perhaps for b_r even if it is below s) and a portion of s is added to B (starting or ending at the intersection point, depending on which part of s is above B).

Finally, we come to the case when both endpoints of s are below B, which leads to the two possibilities illustrated in Figures 2c and 2d. In this case, there may or may not be an intersection and some extra work needs to be done to determine this. Namely, we certainly do not have an intersection when s belongs to the downsloping part of S and p_l is below the upsloping part of B or vice versa, when s has an upward slope and p_r is below the downsloping part of B. However, this is not sufficient to decide whether there is an intersection between s and B. These cases, then, are subsumed by the following simple check. First, we determine whether the slope of s is between the slope of b_l and that of b_r (remember, that slopes uniformly increase from b_l to b_r). Only if it is, there may be an intersection. We then find, via a binary search on the slopes of lines between b_l and b_r, the line b of B that has slope closest to that of s. If this line is not above s (Figure 2c), then we have two intersections which can be found by walking from b in opposite directions, while deleting segments from B. Otherwise, there is still no intersection (Figure 2d). To see that this is indeed a

correct strategy, we remember that if s were to pierce B it must either intersect or "obscure" the line with the closest slope since in the resulting envelope lines must appear in the order from smallest to largest slope.

To complete the description of the modified GCSA heuristic, we need to address one more problem and that is the computation of the merge cost, i.e. the eccentricity of the resulting curve if the two allocation sets were merged. This, however, can be achieved with the same algorithm as above except that no changes should be made to B (i.e., we "simulate" a merge) and we can stop once the lowest point on the envelope has been found (note that this technique cannot be used for the exact algorithm in the previous section for it only handles envelopes obtained by merges and does not handle those obtained by deleting lines).

Let's analyze now the running time of this new GCSA algorithm. We first look at the operation of a single merge step involving the smaller allocation set S with $|S| = n_S$ and the bigger allocation set B with $|B| = n_B$. How many times can a segment of S's envelope intersect B's envelope? The answer is at most twice, since envelopes have parabolic shape. Therefore, only one part of a segment of S or that segment in its entirety can be inserted into B's envelope and since the number of segments in the envelope is at most one more than the size of the allocation set, no more than $n_S + 1$ insertions take place. Therefore, the total cost of insertions per merge step is $O(n_S \log n_B)$. It remains to sum the cardinalities of all such smaller allocation sets S participating in merge steps. This question can be approached from the point of view of how many times, at the most, the same point can belong to the smaller set over the course of all merge steps. This is very similar to the analysis of the disjoint data set union operation and we know that the same point can be merged from a smaller set at most $\log n$ times, for the sizes of the smaller sets it is part of will in the worst case increase as the sequence $1, 2, 4, 8, \ldots$ So, the number of insertions over all merge steps is at most $O(n \log n)$ and with each insertion taking $O(\log n)$ time, the total time is $O(n \log^2 n)$. We still need to remember to account for the deletions taking place during merging, but this is easy for once a line has been removed from an envelope, the point responsible for it will no longer be considered. Hence, the total cost of deletions is only $O(n \log n)$. Also, "simulating" a merge to compute the prospective eccentricity has the same cost as an ordinary merge and we know that only at most two such simulations are needed for every real merge step. Thus, we can perform all $O(n)$ merges in $O(n \log^2 n)$ time. This gives us the following result.

Theorem 3. *The modified GCSA algorithm runs in $O(n \log^2 n)$ time and guarantees the error bounds proven for the original GCSA. Namely that for $n \geq 2k$, the GCSA algorithm with $m = 2k-1$ produces a curve C with eccentricity $\epsilon \leq \epsilon^*$ and with $m = k$ segments achieves eccentricity at most $3\epsilon^*$.*

The above claims regarding the error bounds follow directly from the proofs given in [10], as they carry over verbatim to this modified version of GCSA.

5 A Randomized Algorithm

The main idea of this algorithm is to perform an efficient search on the set of $O(n^2)$ possible eccentricities but, unlike [3], the entire set of eccentricities is not generated explicitly. Instead, only those for which a candidate curve is constructed are computed. This results in $O(\log n)$ candidates on average and $O(n \log^2 n)$ expected running time. We begin by describing the unweighted version of the algorithm and then show how to extend it to handle weights.

The algorithm starts by picking a random pair of points p_i and p_j and computing the eccentricity of the allocation set S_{ij}. This can be done in $O(n)$ time (e.g., using the linear programming algorithm in [11]). Then, using the min-# algorithm of [3], the first candidate curve R_{ij} of size k_{ij} is constructed and compared against the target k. The result of this comparison is to be used to decide about the bounds on the achievable eccentricity. The algorithm, therefore, keeps track of the feasible eccentricity window $\mathcal{E}_f = [\varepsilon_{min}, \varepsilon_{max}]$, which is updated after investigating each candidate curve. This window is initialized to $[0, \infty)$. Now, if $k_{ij} \leq k$, we update the window to $[0, \varepsilon_{ij}]$. While, in the opposite case of $k_{ij} > k$, we know that the eccentricity has to be increased, and so the feasible window becomes $[\varepsilon_{ij}, \infty)$.

Now, to discard all allocation sets whose errors are outside of the feasible eccentricity window, we create a data structure that records for each point p_i of S the number of allocation sets that start at p_i and end at some p_j with errors still in the current feasible window as well as the smallest index l_i and the largest index r_i such that $i \leq l_i \leq j \leq r_i$. For each p_i and a given feasible eccentricity window $\mathcal{E}_f$, we thus have the set $S_i^{\mathcal{E}_f}$ of possible values of j. In this set, $j = l_i$ specifies the index of the closest (in x-direction) point to p_i such that the error of the allocation set $\{p_i, \ldots, p_{l_i}\}$ is at least ε_{min} and, similarly, $j = r_i$ gives the furthest point from p_i with the error of the allocation set $\{p_i, \ldots, p_{r_i}\}$ at most ε_{max}. To see that p_j runs across a contiguous subset of S, we note that the eccentricity of an allocation set S_{ij} is monotonically non-decreasing as i is kept fixed and j is advanced.

Let us now explain how to compute for each p_i the cardinality and bounds l_i, r_i of $S_i^{\mathcal{E}_f}$ as well as how to maintain this information as $\mathcal{E}_f$ changes. After the first candidate curve is generated and $\mathcal{E}_f$ is initialized, we compute $|S_1^{\mathcal{E}_f}|$ and l_1, r_1 by scanning S. We then note that $l_i \leq l_k, r_i \leq r_k$ whenever $i \leq k$. This holds because the y-range of the set $\{p_k, \ldots, p_{l_i}\}$ (possibly empty if $k > l_i$) is subsumed by the y-range of the set $\{p_i, \ldots, p_{l_i}\}$ forcing l_k to be no less than l_i and, similarly, for r_k and r_i. Therefore, it seems that l_2 and r_2 can be found by simply moving ahead the pointers from l_1 and r_1, respectively, if needed. Unfortunately, in order to know when to stop for l_2 and r_2 we need to know the error of the allocation sets that begin at p_2 rather than p_1 for it could be that p_1, which is now removed from consideration, was one of the two points determining the error of $L_1 = \{p_1, \ldots, p_{l_1}\}$ or the two points determining the error of $R_1 = \{p_1, \ldots, p_{r_1}\}$. This necessitates the creation of two priority queues, such as min-max heaps, to keep track of the lowest and highest points in the

two allocation sets L_i, R_i as they are being determined for each p_i. Then, in the case of p_2, we set $L_2 = L_1, R_2 = R_1$ and then remove p_1 from the heaps for each of the sets. Then, we start adding points to L_2 beginning with p_{l_1+1} and stop after having added the first point that had caused the error of L_2 to exceed or become equal to ε_{min}. Similarly, we add points to R_2 until adding the next point would make the error of R_2 become greater than ε_{max}. We remember the index of the last point added to L_2 as l_2 and that of the last point added to R_2 as r_2. All subsequent bounds for $S_i^{\mathcal{E}_f}, 2 \leq i \leq n$, can be found by advancing these two pointers, each making at most one full pass through S. Finally, computing the size of $S_i^{\mathcal{E}_f}$ is trivial as it is just $|r_i - l_i + 1|$. We note that every point is added to each of the two queues exactly once and is removed at most once as we compute all values of i_l, i_r for a given eccentricity window $\mathcal{E}_f$.

Thus, every time $\mathcal{E}_f$ changes, recomputing the bounds and cardinality information takes only $O(n \log n)$ time since it only involves $O(n)$ heap operations. Hence, the key to good performance becomes reducing the (expected) number of changes to the feasible eccentricity window that are necessary to process before the optimal eccentricity is found. This goal we achieve through randomization as we shall describe next.

After the initial step has determined $\mathcal{E}_f$ and the bounds and cardinalities of each of the sets $S_i^{\mathcal{E}_f}$ have been computed, we pick a pair of points that enclose an allocation set with error in the feasible eccentricity window at random from the set of all such possible pairs. In order to do this, and have a uniform distribution of probabilities, for each $2 \leq i \leq n$ we sum up the cardinalities of the sets $S_k^{\mathcal{E}_f}$ for all $k \leq i$ and store this number for p_i, i.e. we have

$$K_i = \sum_{k=1}^{i} \left| S_k^{\mathcal{E}_f} \right|.$$

Clearly, these can be computed in one scan of the array since K_i is just the sum of K_{i-1} and the cardinality of $S_i^{\mathcal{E}_f}$. Then, we can generate a pseudo-random number x between 1 and K_n and identify the unique pair (p_i, p_j) corresponding to this index (we just search for x in the array of K_i's, find the smallest i_0 such that $K_{i_0} \geq x$, and then find the unique j from $S_{i_0}^{\mathcal{E}_f}$.

This way we make sure that each pair is selected with the same probability but only from the set of those pairs that already fulfill the criteria for the error of its allocation set. Thus, we can expect that on average picking a new pair will reduce the number of pairs in the feasible eccentricity window roughly in half and so, our search has an expected logarithmic number of steps in the size of the set of possible eccentricities, that is, $O(\log(n^2)) = O(\log n)$. Since after each pair is picked an $O(n \log n)$ time is spent updating the auxiliary arrays described above and constructing a candidate curve, the total expected running time of this randomized algorithm is $O(n \log^2 n)$.

Now, notice that even though the discussion so far focused on the unweighted case only, our algorithm can be easily adapted to the weighted case. First, we observe that it is still true in the presence of weights that $l_i \leq l_k, r_i \leq r_k$ for

any two points p_i, p_k such that $i \leq k$. Clearly, not just the y-range but the weighted error range of the set $\{p_k, \ldots, p_{l_i}\}$ is subsumed by the weighted error range of $\{p_i, \ldots, p_{l_i}\}$ because the lowest point on the upper envelope of a set of cost lines can be no lower than the lowest point on the upper envelope of its subset. Consequently, instead of min-max heaps to keep track of the errors of L_i and R_i we would have to maintain upper envelopes and we can do so again using the algorithm from [13]. Each individual update of that structure takes $O(\log^2 n)$ and so one full pass through the array to update the pointers l_i, r_i for all i takes $O(n \log^2 n)$. Hence, the total time is $O(n \log^3 n)$ as there are still $O(\log n)$ candidate curves to construct.

References

1. Aronov, B., Asano, T., Katoh, N., Mehlhorn, K., Tokuyama, T.: Polyline fitting of planar points under min-sum criteria. International Journal of Computational Geometry and Applications. **16** (2006) 97–116
2. Brodal, G., Jacob, R.: Dynamic planar convex hull. Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science. (2002) 617–626
3. Díaz-Báñez, J.M., Mesa, J.A.: Fitting rectilinear polygonal curves to a set of points in the plane. European Journal of Operations Research. **130** (2001) 214–222
4. Eu, D., Toussaint, G.: On approximating polygonal curves in two and three dimensions. CVGIP: Graphical Models and Image Processing. **56**(3) (1994) 231–246
5. Goodrich, M.: Efficient piecewise-linear function approximation using the uniform metric. Discrete and Computational Geometry. **14** (1995) 445–462
6. Hakimi, S.L., Schmeichel, E.F.: Fitting polygonal functions to a set of points in the plane. CVGIP: Graphical Models and Image Processing. **53**(2) (1991) 132–136
7. Hershberger, J., Suri, S.: Off-line maintenance of planar configurations. Journal of Algorithms. **21** (1996) 453–475
8. Houle, M., Imai, H., Imai, K., Robert, J.-M., Yamamoto, P.: Orthogonal weighted linear L_1 and L_∞ approximation and applications. Discrete Applied Mathematics. **43**(3) (1993) 217–232
9. Imai, H., Iri, M.: Computational-geometric methods for polygonal approximations of a curve. Computer Vision, Graphics and Image Processing. **36**(1) (1986) 31–41
10. Mayster, Y., Lopez, M.A.: Rectilinear approximation of a set of points in the plane. LATIN 2006: Theoretical Informatics. (2006) 715–726
11. Megiddo, N.: Linear programming in linear time when the dimension is fixed. Journal of ACM. **31**(1) (1984) 114–127
12. Melkman, A., O'Rourke, J.: On polygonal chain approximation. In: Toussaint, G.T. (ed.): Computational Morphology. North-Holland, Amsterdam, Netherlands (1988) 87–95
13. Overmars, M.H., van Leeuwen, J.: Maintenance of configurations in the plane. Journal of Computer and System Sciences. **23**(2) (1981) 166–204
14. Seidel, R.: Linear programming and convex hulls made easy. SCG '90: Proceedings of the Sixth Annual Symposium on Computational Geometry. (1990) 211–215
15. Wang, D.P.: A new algorithms for fitting a rectilinear x-monotone curve to a set of points in the plane. Pattern Recognition Letters. **23** (2002) 329–334
16. Yamamoto, P., Kato, K., Imai, K., Imai, H.: Algorithms for vertical and orthogonal L_1 linear approximation of points. Proceedings of the 4th Annual Symposium on Computational Geometry. (1988) 352–361