Fitting a step function to a point set

Hervé Fournier * Antoine Vigneron †

June 12, 2009

Abstract

We consider the problem of fitting a step function to a set of points. More precisely, given an integer k and a set P of n points in the plane, our goal is to find a step function f with k steps that minimizes the maximum vertical distance between f and all the points in P. We first give an optimal $\Theta(n \log n)$ algorithm for the general case. In the special case where the points in P are given in sorted order according to their x-coordinates, we give an optimal $\Theta(n)$ time algorithm. Then, we show how to solve the weighted version of this problem in time $O(n \log^4 n)$. Finally, we give an $O(nh^2 \log n)$ algorithm for the case where h outliers are allowed. The running time of all our algorithms is independent of k.

Keywords: Algorithm Design; Optimization; Computational Geometry; Shape Fitting; Histogram

1 Introduction

In this paper, we consider the problem of fitting a step function to a point set in $\mathbb{R}^2$. (See Figure 1 for an example.) For a given number of steps k, we find the step function whose maximum vertical distance to a point set P is minimized.

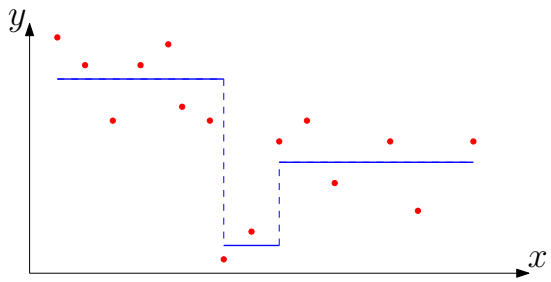

Figure 1: A set of points in $\mathbb{R}^2$ and an optimal approximation by a 3-step function.

The motivation for this work is to find a concise representation of a large set of points by a step function with few steps. This type of representation of point-sets by step functions (also called *histograms*) is used in Database Management Systems, for *query optimization*: there can be many different ways of answering

*Laboratoire PRiSM, CNRS UMR 8144 and Université de Versailles Saint-Quentin-en-Yvelines, 45 avenue des États-Unis, 78035 Versailles, France. Email: herve.fournier@prism.uvsq.fr.

†INRA, UR 341 Mathématiques et Informatique Appliquées, 78350 Jouy-en-Josas, France. Email: antoine.vigneron@jouy.inra.fr.

a complex query involving several attributes of the data, and query optimization consists in predicting the fastest way to answer a query, before this query is performed. This prediction is done using some statistics on the data, which often consists of histograms. Several types of histograms have been used in databases [16]; the histograms that correspond to our optimal step functions are called *maximum error histograms*, and have recently been studied in the database community [7, 13, 17].

We give optimal algorithms for computing the optimal step function in our model; in other words, we give optimal algorithms for computing maximum error histograms. We also give efficient algorithms for two generalizations. First, we consider the more general case where each point is weighted, and its contribution to the distance computation is multiplied by this weight. Second, we introduce a generalization of our problem to the case where outliers are allowed; that is, we allow our algorithm to ignore h input points, where h is an input parameter. The motivation is to make the algorithm more robust to noise in the input point set.

1.1 Problem formulations

A function $f : [a, b) \to \mathbb{R}$ is called a *k-step function* if there exists a real sequence $a = a_0 < a_1 < \cdots < a_k = b$ such that the restriction of f to each interval $[a_i, a_{i+1})$ is a constant. We use $\mathrm{d}(p, f)$ to denote the vertical distance between a point $p \in \mathbb{R}^2$ and f; in other words when $p = (x, y)$, we have $\mathrm{d}(p, f) = |f(x) - y|$. Let P denote a set of n points in $\mathbb{R}^2$. We define the distance $\mathrm{d}(P, f)$ between P and a step function f as follows:

$$\mathrm{d}(P, f) = \max\{\mathrm{d}(p, f) \mid p \in P\}.$$

We will consider the following three problems. The first one is the problem that we already mentioned and the second one is the weighted version. The third one is a generalization where we allow h outliers, which means that we allow a given number h of points from P to be far from f.

- MIN-DIST: Given an integer $k > 0$ and a set P of n points in $\mathbb{R}^2$, find a k-step function f^* such that $\varepsilon^* = \mathrm{d}(P, f^*)$ is minimized.
- weighted-MIN-DIST: a generalization of MIN-DIST where we are given a set of weighted points, and the distance between f and a point p with weight μ is $\mu \cdot \mathrm{d}(p, f)$. The goal is still to minimize the maximum of these distances for a given k.
- OUTLIER: Given $P \subseteq \mathbb{R}^2$, an integer $k > 0$ and $h \in \mathbb{N}$, find a k-step function f and a subset $P' \subseteq P$ such that $|P'| \leqslant h$ and $\mathrm{d}(P \setminus P', f)$ is minimized.

1.2 Previous work

Algorithms for fitting a polyline to a dataset have been extensively studied. For instance, Goodrich [12] gave an $O(n \log n)$ time algorithm for fitting an x-monotone polyline with k edges to a set of points, under the same criterion as ours: minimizing the maximum vertical distance between the input point set and the polyline. Goodrich's work was motivated by applications to geographic information systems and image processing, where one wants to simplify a curve using a polyline with few edges. References to related work can be found in Goodrich's article [12]. There has also been extensive research in algorithms for fitting a step function to a point set under criteria that are different from ours: for instance, minimizing the sum of the squared vertical distances. The interested reader can find references to these results in Guha and Shim's article [13]. We are not aware of any work for fitting a step function or a polyline that explicitly handles outliers in the same way as we do, but it has been considered for other shape fitting problems. See

for instance the work of Har-Peled et al. [1, 14] on the minimum width annulus and related problems, or the recent paper by Atanassov et al. [4] on outliers removal minimizing parameters such as diameter, perimeter of parallel-axis bounding box, or area of the convex hull.

We now mention results that are more directly related to this paper. Diáz-Báñez and Mesa [20] showed that the dual problem where we want to minimize the number of steps k for a fixed error bound ε can be solved in $O(n)$ time by a greedy approach. This algorithm can be used as a decision algorithm for the MIN-DIST problem, and since the optimal distance ε^* in the MIN-DIST problem is realized by half the difference between two y-coordinates in P, then by sorting these $O(n^2)$ values and applying binary search, Diáz-Báñez and Mesa [20] obtained an $O(n^2 \log n)$ algorithm. This result was improved to $O(n^2)$ by Wang [21], and then by Mayster and Lopez [19] who gave an $O(\min(n^2, nk \log n))$ algorithm for the MIN-DIST problem. Recently, Guha and Shim [13] found an algorithm with running time $O(n+k^2 \log^3 n)$ when the input points are given in sorted order, by increasing x-coordinate. Their algorithm also applies to a model of maximum relative error and to the case where points are weighted. For the latter case, the bound becomes $O(n \log n + k^2 \log^6 n)$; for large k, this was later improved by Lopez and Mayster [18] who gave an algorithm with running time $O(n^2)$. The algorithm by Lopez and Mayster also improves the space requirement of Guha and Shim's algorithm from $O(n \log n)$ to $O(n)$.

Finally, we mention two recent articles from the database community on maximum error histograms. (Which, in our terminology, is the MIN-DIST problem.) Karras et al. [17] give an $O(nL)$-time algorithm for MIN-DIST with sorted input, where in the worst case, L is the number of bits used for representing each coordinate of the input points. They achieve it using the greedy decision algorithm by Diáz-Báñez and Mesa [20], together with binary search. Buragohain et al. [7] gave an efficient algorithm for computing maximum error histograms over data streams.

1.3 Our results

In Section 2, we give a simple $O(n \log n)$ time algorithm for MIN-DIST. We achieve it by combining the technique of Frederickson and Johnson [10] for searching a sorted matrix, with the linear-time decision algorithm by Diáz-Báñez and Mesa [20]. When the input points are not given in sorted order, our algorithm is optimal, and it improves on all previous algorithms [13, 19, 20, 21] in the worst case. In particular, our algorithm is the first one to be truly subquadratic, as previous algorithms run in time $\Omega(n^2)$, $\Omega(k^2)$, or $\Omega(nk)$ in the worst case. It is particularly significant for applications to databases, where n and k could be large enough to make a quadratic running time unacceptable.

In Section 3, we give a more general formulation of Frederickson's path partitioning algorithm [9]. In Section 4, we combine this technique with efficient data structures [6, 11] for reporting range maxima, and we obtain a linear time algorithm for the MIN-DIST problem where the input points are given in sorted order, by increasing x-coordinates. It is the first optimal algorithm for this problem.

Our approach also gives an $O(n \log^4 n)$ algorithm for weighted-MIN-DIST. (See Section 5.) It is the first truly subquadratic algorithm for this problem, and it improves on the $O(n \log n + k^2 \log^6 n)$ algorithm of Guha and Shim [13] when $k = \omega(\sqrt{n}/\log n)$.

Finally, we give an $O(nh^2 \log n)$ algorithm for the OUTLIER problem, where up to h input points can be ignored. It improves on the obvious $O(n^2 h^2)$ dynamic programming approach.

2 A simple optimal algorithm for MIN-DIST with unsorted input

In this section we present our algorithm for the MIN-DIST problem. We first present the sorted matrix searching technique [9, 10], then we give the greedy, linear time decision algorithm for sorted input (which is essentially the algorithm by Diáz-Báñez and Mesa [20] for the dual problem), and then we combine these two results to get an $O(n \log n)$ time optimization algorithm.

2.1 Searching in a sorted matrix

In this section we quickly describe the technique of Frederickson and Johnson for searching in a sorted matrix [9, 10]. Suppose we are given an $n \times n$ sorted matrix M—by sorted, we mean that $i \leqslant i'$ and $j \leqslant j'$ implies that $M_{i,j} \leqslant M_{i',j'}$. We assume that we can access in constant time the value of each element $M_{i,j}$.

Let $g : \mathbb{R} \to \{\text{TRUE}, \text{FALSE}\}$ be a monotone function in the sense that if $x < y$ and $g(x) = \text{TRUE}$, then $g(y) = \text{TRUE}$. Our goal is to find the smallest element $M_{i,j}$ of the matrix M such that $g(M_{i,j}) = \text{TRUE}$; we call this problem the *optimization problem*. The *decision problem* is to compute the mapping g: given $x \in \mathbb{R}$, compute the value of $g(x)$. If we assume that the decision problem can be solved in time D, then the optimization problem can be trivially solved in time $O(n^2 \log n + D \log n)$ by sorting $\{M_{i,j} \mid 1 \leqslant i, j \leqslant n\}$ and then performing a binary search over these values. The algorithm by Diáz-Báñez and Mesa [20] applies this approach using the decision algorithm presented in Section 2.2, whose running time is $D = O(n)$. The technique of Frederickson and Johnson improves the time bound of this optimization technique to $O(n + D \log n)$ as follows.

The algorithm maintains a collection $\mathcal{M}$ of submatrices of M, and initially we set $\mathcal{M} = \{M\}$. We assume that n is a power of 2; if not, we reduce to this case by padding copies of $M_{n,n}$ to the input until its size is a power of 2. At each step, we partition each matrix in $\mathcal{M}$ into four equal-size square matrices. We do not maintain these matrices explicitly (which would require quadratic time): we only keep the range of indices in the original matrix M that corresponds to each submatrix. Then about half of these matrices are discarded. We repeat this process until the sum of the numbers of elements of these submatrices is $O(1)$, and thus we can find the optimal element by brute force.

We still need to explain how submatrices are discarded. Without loss of generality, we assume that all elements of M are distinct (if not, we break ties arbitrarily). For each submatrix in $\mathcal{M}$, we compute the smallest element (which of course is at the upper left corner). Then we compute the median $\lambda_{\min}$ of these smallest elements. We also compute the median $\lambda_{\max}$ of the largest elements. Suppose now that $g(\lambda_{\min}) = \text{TRUE}$. Then clearly, we can discard all the submatrices whose smallest element is larger than $\lambda_{\min}$. Otherwise, if $g(\lambda_{\min}) = \text{FALSE}$, we discard all the submatrices whose largest element is smaller than $\lambda_{\min}$. We handle $\lambda_{\max}$ in a similar way: if $g(\lambda_{\max}) = \text{TRUE}$, we discard all submatrices whose smallest element is larger than $\lambda_{\max}$, and if $g(\lambda_{\max}) = \text{FALSE}$, we discard all submatrices whose largest element is smaller than $\lambda_{\max}$. It can be shown that together, these two steps allow to discard almost one half of the matrices. For more details, we refer to the papers by Frederickson and Johnson [9, 10] and the survey by Agarwal and Sharir [2, Section 3.3].

2.2 A decision algorithm for sorted input

In this section, we describe the linear-time decision algorithm by Díaz-Báñez and Mesa [20]. We assume that P is given as a set of points $p_i = (x_i, y_i)$ for $1 \leqslant i \leqslant n$ sorted from left to right: we assume that for all i, we have $x_i < x_{i+1}$. (To simplify the exposition, we assume that no two points have same x-coordinate.)

We denote $P[i,j] = \{p_i, \dots p_j\}$, $Y[i,j] = \{y_i, \dots y_j\}$ and $\Delta[i,j] = (\max Y[i,j] - \min Y[i,j])/2$. In other words, $\Delta[i,j]$ is the distance between $P[i,j]$ and the closest 1-step function.

Our decision algorithm takes as input the point set P, an integer k, and $\varepsilon \geqslant 0$. It returns TRUE if there exists a k-step function f such that $\mathrm{d}(f,P) \leqslant \varepsilon$, and returns FALSE otherwise. We define a function $\mathrm{next}_\varepsilon$ as follows: if $\Delta[i,n] \leqslant \varepsilon$, then $\mathrm{next}_\varepsilon(i) = n+1$ and otherwise, $\mathrm{next}_\varepsilon(i)$ is the smallest integer $j > i$ such that $\Delta[i,j] > \varepsilon$. We first observe that for all i, we can compute $\mathrm{next}_\varepsilon(i)$ in time $O(\mathrm{next}_\varepsilon(i) - i)$ by traversing $P[i,n]$ from left to right with a pointer i', while maintaining the value of $\max Y[i,i']$ and $\min Y[i,i']$. We are now ready to give the decision algorithm:

Algorithm DECIDE(P, k, ε)
1. $i_1 \leftarrow 1$.
2. **for** $j \leftarrow 1$ **to** k
3. **do** $i_{j+1} \leftarrow \mathrm{next}_\varepsilon(i_j)$.
4. **if** $i_{j+1} = n+1$
5. **then return** TRUE.
6. **return** FALSE.

From our observation that $\mathrm{next}_\varepsilon(i)$ can be computed in time $O(\mathrm{next}_\varepsilon(i) - i)$, we can see that algorithm DECIDE runs in $O(n)$ time. A fairly simple argument shows that this algorithm is correct; the interested reader can find a detailed proof in the article by Díaz-Báñez and Mesa [20]. Thus, we have the following result:

Lemma 1 *Given a sorted point set P, an integer k, and $\varepsilon > 0$, the algorithm* DECIDE(P, k, ε) *decides in $O(n)$ time whether there exists a k-step function f such that $\mathrm{d}(f,P) \leqslant \varepsilon$.*

2.3 Optimization algorithm

We denote by E the set of half-differences between y-coordinates in P:

$$E = \left\{ \frac{y - y'}{2} \mid (x,y) \in P \text{ and } (x',y') \in P \right\}$$

It is easy to see that the error ε^* of the solution to the MIN-DIST problem is in this set. The MIN-DIST problem can be solved by sorting E in $O(n^2 \log n)$ time, and then looking for the optimal value ε^* using the decision algorithm and binary search.

In order to avoid a quadratic running time, we will apply the sorted matrix searching technique (Section 2.1). This is made possible by the fact that E can be written as follows:

$$E = Y + (-Y) \text{ where } Y = \left\{ \frac{y}{2} \mid (x,y) \in P \right\}.$$

Thus we first sort Y and obtain a non-decreasing sequence (b_i) such that $Y = \{b_1 \leqslant b_2 \dots \leqslant b_n\}$. Then we represent the elements of E as the set of elements of a sorted matrix M defined by $M_{i,j} = \frac{1}{2}(b_i - b_{n+1-j})$.

So our optimization algorithm works as follows. We first place in a sorted array the elements of Y, which takes $O(n \log n)$ time. It allows us to compute elements of M in constant time. Then we sort P according to the x-coordinates, which, by Lemma 1, allows us to run the decision algorithm in $O(n)$ time for any value of ε. So, applying the sorted matrix searching technique (see Section 2.1), we obtain the following result:

Theorem 2 *The* MIN-DIST *problem can be solved in $O(n \log n)$ time.*

A simple reduction from sorting shows that it is optimal when the input point-set P is not given in sorted order: given the input $(x_1, \ldots, x_n)$, take $P = \{(x_i, x_i) \mid 1 \leqslant i \leqslant n\}$ and $k = n$. The sequence of y-values taken by an optimal k-step function corresponds to $(x_1, \ldots, x_n)$ in sorted order.

3 Frederickson's algorithm for Min-Max partitioning

Frederickson [9] gave a linear time algorithm for the following path partitioning problem: given an integer $k > 0$ and n positive real numbers $\omega_1, \ldots, \omega_n$, compute a partition of $\{1, \ldots, n\}$ into k intervals $I_1, \ldots, I_k$ such that $\max_{j \in \{1,\ldots,k\}} \sum_{i \in I_j} \omega_i$ is minimized.

Let us first formulate Frederickson's min-max partition problem in a more general setting. Let Σ be a set (not necessarily finite); we shall call it the alphabet. We denote by Σ^* the set of words over Σ, and the empty word is denoted by e. For $v, w \in \Sigma^*$, vw denotes the concatenation of v and w. Assume that we have a mapping $\theta : \Sigma^* \to \mathbb{R}^+$ such that $\theta(e) = 0$. We are interested in the following problem: given $w \in \Sigma^*$, compute a factorization $w = w_1 w_2 \ldots w_k$ (where $w_i \in \Sigma^*$) such that $\max_{i \in \{1,\ldots,k\}} \theta(w_i)$ is minimized. We shall call this problem MIN-MAX PARTITION(θ). Note that the problem of Frederickson corresponds to MIN-MAX PARTITION(S) with $\Sigma = \mathbb{R}^+$ and $S(\omega_i \ldots \omega_j) = \omega_i + \ldots + \omega_j$. The following result gives sufficient conditions on θ which allow to apply Frederickson's technique to solve MIN-MAX PARTITION(θ).

Theorem 3 *Let Σ be a set, and $\theta : \Sigma^* \to \mathbb{R}^+$ be a mapping such that $\theta(e) = 0$. Suppose that θ has the following properties:*

(i) θ is non-decreasing, that is, $\theta(v) \leqslant \theta(uvw)$ for all $u, v, w \in \Sigma^*$.

(ii) We can preprocess $a_1 \ldots a_n \in \Sigma^n$ in time $\pi(n)$ so that, given any query (i, j), we can compute $\theta(a_i \ldots a_j)$ in time $\kappa(n)$.

Then MIN-MAX PARTITION*(θ) can be solved in time $O(\pi(n) + n\kappa(n))$.*

To see why this theorem holds, one just has to reformulate the proof of Frederickson [9] using our more general framework. As it would be quite technical, we only reprove what corresponds to the $O(n \log\log n)$ algorithm given by Frederickson, which gives an $O(\pi(n) + n(\log\log n)\kappa(n))$ time bound in our case. The rest of the proof in Frederickson's paper extends directly to our case, which proves Theorem 3.

To simplify the presentation, we assume that $\kappa(n) = O(1)$ in the remainder of this section. The lemma below shows that conditions (i) and (ii) yield a linear time algorithm for the decision problem.

Lemma 4 (Naive decision algorithm) *Under the hypothesis of Theorem 3, and after the preprocessing step corresponding to condition (ii), there exists a linear time algorithm for the decision problem relative to* MIN-MAX PARTITION*(θ).*

Proof: Once the preprocessing on θ has been done, a greedy algorithm for the decision problem can be achieved by a single sweep from left to right. It is essentially the same algorithm as the decision algorithm for MIN-DIST presented in Section 2.2, and the proof carries over to this case. □

□

Now we still need to design an algorithm to compute the optimal value ε^*. The algorithm to compute this value relies on the technique of matrix searching. We recall the main result in its general version as it appears in Frederickson's article [9], but restricted to the case of square matrices of the same size.

Theorem 5 (Matrix Searching) *Let $\mathcal{M}$ be a collection of N sorted matrices $\{M_1, \ldots, M_N\}$ in which matrix M_j is of size $m \times m$. Let s be a non-negative integer (called the stopping count). The number of calls to the decision algorithm that are needed by the matrix searching algorithm to discard all but at most s of the elements is $O(\max\{\log m, \log(Nm/(s+1))\})$, and the total time of the Matrix Searching algorithm exclusive of the calls to the decision algorithm is $O(Nm)$.*

Given an input $w = a_1 \ldots a_n \in \Sigma^n$ and $k > 0$ (both fixed from now on), we define $\theta_{i,j} = \theta(a_i \ldots a_j)$ for all $i \leqslant j$. Let us define the $n \times n$ matrix M by $M_{i,j} = \theta_{n+1-i,j}$ when $n+1 \leqslant i+j$ and $M_{i,j} = 0$ otherwise. Because of property (i), M is a sorted matrix: it is non-decreasing along each line and each column. Let ε^* be the optimal value of MIN-MAX PARTITION(θ) on input (w, k). Of course $\varepsilon^* \in M$. Condition (ii) allows to query any entry of the matrix M defined above in time $O(1)$ with preprocessing time $\pi(n)$ on the input $(a_1, \ldots, a_n)$. Note that we never do any other preprocessing of this type: all matrices occurring in the algorithm are seen as a product of two intervals of $\{1, \ldots, n\}$, and querying these matrices is performed via a query on the big matrix M. In the next two lemmas, we assume that the preprocessing corresponding to condition (ii) of Theorem 3 has been done, and we only consider the running times after this step.

The main idea to obtain the $O(n \log\log n)$-time optimization algorithm is to get a faster decision algorithm by doing some additional preprocessing on w. Then ε^* is computed by applying the technique of matrix searching on M, using this improved decision algorithm.

The preprocessing is performed as follows. Let r be an integer (to be chosen later). The input w is divided into $\lceil n/r \rceil$ factors v_i of length r, thus we have $w = v_1 \ldots v_{n/r}$. The family of sorted matrices $\mathcal{M} = \{M_1, \ldots, M_{n/r}\}$ is created, where M_i contains the values of θ over v_i. Then a first phase of matrix searching is performed on $\mathcal{M}$ with stopping count n/r^2. While doing this search, the decision algorithm is run for several values of ε. We denote by λ_1 the largest such value of ε that is not feasible, and we denote by λ_2 the smallest one that is feasible. Hence, we have $\lambda_1 < \varepsilon^* \leqslant \lambda_2$. At most n/r^2 factors v_p have (at least) an element $\theta_{i,j}$ of their matrix M_p lying in $[\lambda_1, \lambda_2]$; we call these factors *active*. The other factors are called non-active.

We shall carry out some preprocessing on the non-active factors. Let us consider a factor $v_i = a_p \ldots a_q$. For $t \in \{p, \ldots, q\}$, we define $\mathrm{ncut}(t)$ to be the minimum index ℓ such that there exists a partition $a_t \ldots a_q = u_1 \ldots u_{\ell+1}$ such that $\max_i \theta(u_i) \leqslant \lambda_1$. Moreover, we define $\mathrm{rem}(t)$ to be the maximum of j where $u_{\ell+1} = a_j \ldots a_q$ over all these partitions. Notice that these two parameters $\mathrm{ncut}(t)$ and $\mathrm{rem}(t)$ are the ones obtained by applying the greedy decision algorithm given in Lemma 4 on $a_p \ldots a_q$ and λ_1. The next lemma explains how to compute these values efficiently.

Lemma 6 (Preprocessing factors) *Given an interval of $\{1, \ldots, n\}$ of length r, computing $\mathrm{rem}(t)$ and $\mathrm{ncut}(t)$ for all $t \in I$ can be done in time $O(r)$.*

Proof: For $t \in I$, we define $\mathrm{next}_{\lambda_1}(t)$ to be the smallest t' such that $\theta_{t,t'} > \lambda_1$. All values of $\mathrm{next}_{\lambda_1}(t)$ can be computed in time $O(r)$ by scanning from left to right, while maintaining a pointers to t and a pointer to $\mathrm{next}_{\lambda_1}(t)$. Then, $\mathrm{ncut}(t)$ and $\mathrm{rem}(t)$ can be computed by a single scan from right to left. At each step of this scan, we compute $\mathrm{ncut}(t)$ and $\mathrm{rem}(t)$ in constant time by accessing $t' := \mathrm{next}_{\lambda_1}(t)$, and then using the already computed values $\mathrm{ncut}(t')$ and $\mathrm{rem}(t')$. □

□

Once this preprocessing has been done, we obtain a sublinear decision algorithm.

Lemma 7 (Sublinear decision algorithm) *After the preprocessing of Lemma 6 has been done, we have an $O((n/r)\log r)$ time algorithm to solve the decision problem corresponding to* MIN-MAX PARTITION*(θ).*

Proof: We can assume that $\lambda_1 < \varepsilon < \lambda_2$, because if $\varepsilon \leqslant \lambda_1$ we can immediately return FALSE, and if $\varepsilon \geqslant \lambda_2$ we can return TRUE. The idea is to implement the greedy approach faster, by jumping inside non-active factors; on the other hand, we still operate by brute force within active factors. So let's assume that the naive, greedy approach yields a factorization $w = u_1 \dots u_q$.

Consider the maximum sequence $u_i \dots u_j$ (possibly the empty word) which entirely lies in the p-th subsequence v_p of w. Hence we can write $v_p = uu_i \dots u_j u'$. Assume that v_p is non-active. For all $i \leqslant i' \leqslant j$, we have $\theta(u_{i'}) \leqslant \varepsilon$, so by definition of non-active factors, we have $\theta(u_{i'}) < \lambda_1$. It follows that, if a_t is the first letter in u_i, we have that $j - i = \mathrm{ncut}(t)$ and u' starts at index $\mathrm{rem}(t)$.

As a result, we can implement the greedy algorithm in $O(\log r)$ time within a non-active factor v_p. First we find by binary search the index t of first cut inside v_p, using the index of the previous cut. Then we find the number of cuts needed inside v_p using $\mathrm{ncut}(t)$. Finally, we know that the last cut inside v_p is at $\mathrm{rem}(t)$.

There are at most n/r^2 active factors, so our improved greedy algorithm spends a total of $O(n/r)$ time on them. On the other hand, it spends time $O(\log r)$ on each non-active interval, and there are less than n/r of them. It yields the desired time bound. □

□

We can now describe the whole $O(\pi(n) + n \log\log n)$-time algorithm. First the preprocessing corresponding to condition (ii) is performed on the input w. The sorted matrix M is created. Then we choose $r = \lfloor \log n \rfloor$ and we create the family of sorted matrices $\mathcal{M}$ corresponding to the n/r factors $v_1, \dots, v_{n/r}$ of length r of w. A first phase of matrix searching is performed on $\mathcal{M}$ using the naive decision algorithm (Lemma 4) and stopping count n/r^2; it provides new bounds $\lambda_1 < \varepsilon^* \leqslant \lambda_2$. Then we perform the preprocessing (Lemma 6) with respect to λ_1 on all non-active factors. The last step of the algorithm consists in a matrix search on the big matrix M with stopping count $O(1)$, using the sublinear decision algorithm (Lemma 7). It gives the optimal value ε^*. It is easy to check that the running time of this algorithm is $O(\pi(n) + n \log\log n)$, thanks to our improved decision algorithm that runs in time $O(n(\log\log n)/\log n)$.

Frederickson's $O(\pi(n) + n)$-time algorithm is based on clever pruning techniques, and cutting the intervals recursively (with finely tuned parameters) to obtain faster decision algorithms at each step. A careful reading of the proof of Frederickson shows that conditions (i) and (ii) are sufficient to be able to apply these techniques exactly in the same way. We do not rewrite this proof here; the interested reader can check the original paper of Frederickson [9] to complete the proof of Theorem 3.

4 A linear time algorithm for MIN-DIST with sorted input

In this section, we give an optimal, linear-time algorithm for the MIN-DIST problem with sorted input. We achieve it by combining our reformulation of Frederickson's technique (Theorem 3) with efficient data structures for range reporting.

The input of sorted-MIN-DIST is an integer k, and a set $P = \{(x_1, y_1), \dots, (x_n, y_n)\}$ of n points in the plane. These points are given as a sequence, sorted according to their x-coordinates, so we have $x_1 \leqslant \dots \leqslant x_n$. This problem can be reduced in linear time to the case where at most two points have the same x-coordinate: we only need to rewrite the input under the form $((x_1, y_1, y'_1), \dots, (x_n, y_n, y'_n))$

with $x_1 < \ldots < x_n$ and $y_i \leqslant y'_i$ for all i. The x-coordinates obviously play no role in this problem, so we are left with an input which is a positive integer k, and a sequence of intervals of the real line $([y_1, y'_1], \ldots, [y_n, y'_n])$. The sorted-MIN-DIST problem corresponds to MIN-MAX PARTITION(Δ) with $\Sigma = \{(a, b) \in \mathbb{R}^2 \mid a \leqslant b\}$ and

$$\Delta([z_1, z'_1], \ldots, [z_p, z'_p]) := (\max(z'_1, \ldots, z'_p) - \min(z_1, \ldots, z_p))/2.$$

The mapping Δ is obviously non-decreasing, that is, it satisfies property (i) of Theorem 3. Let us now show that it satisfies property (ii). Given an input $([y_1, y'_1], \ldots, [y_n, y'_n])$ and k, we define $\Delta_{i,j} := \Delta([y_i, y'_i], \ldots, [y_j, y'_j])$. After $O(n)$ preprocessing time, we need to be able to compute $\Delta_{i,j}$ in time $O(1)$ for any given i and j. To this end, we use an efficient algorithm for the range maxima problem by Bender and Farach-Colton [6] (which simplifies previous algorithms [11, 15]). More precisely, given a sequence of numbers $(a_1, \ldots, a_n)$, it allows us to answer any query $(i, j) \mapsto \max(a_i, \ldots, a_j)$ in constant time, after preprocessing time $O(n)$. After preprocessing $(y'_1, \ldots, y'_n)$ in this way, we can query $\max(y'_i, \ldots, y'_j)$ in time $O(1)$. In the same way, we can preprocess $(y_1, \ldots, y_n)$ in linear time with respect to the query $(i, j) \mapsto \min(y_i, \ldots, y_j)$. It allows us to compute $\max(y'_i, \ldots, y'_j)$ and $\min(y_i, \ldots, y_j)$, and thus $\Delta_{i,j}$, in $O(1)$ time per query after preprocessing time $O(n)$. Thus Δ has property (ii) with $\pi(n) = O(n)$ and $\kappa(n) = O(1)$. So by Theorem 3, we have proved the following:

Theorem 8 *The sorted-*MIN-DIST *problem can be solved in linear time.*

This algorithm is optimal, since no algorithm can solve this optimization problem without reading the y-coordinates of all the input points. Suppose indeed that an algorithm returns a function f on the input (P, k) without accessing the y-coordinate of the point p_{i_0}. Consider $\tilde{P}$ to be same as P except that the y-coordinate of p_{i_0} is replaced with z. This algorithm will return the same function f on input $(\tilde{P}, k)$, but f is not optimal anymore if z is large enough. It gives an $\Omega(n)$ lower bound on this problem and shows the optimality of our linear time algorithm.

5 MIN-DIST with weighted inputs

In this section, we give a near-linear time algorithm for the weighted version of the MIN-DIST problem. Again, it is an application of Theorem 3. We also use an efficient data structure by Guha and Shim [13] for the related query problem.

Let us recall the weighted MIN-DIST problem defined by Guha and Shim [13]—where it is called Maximum Error Histogram. Given a collection of points in the plane $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ with positive weights $\mu_1, \ldots, \mu_n$ and an integer $k > 0$, compute a k-step function f such that $\max_{1 \leqslant i \leqslant n} \mu_i |f(x_i) - y_i|$ is minimized. Guha and Shim [13] give an algorithm to solve weighted-MIN-DIST in time $O(n \log n + k^2 \log^6 n)$ and space $O(n \log n)$. We obtain the following alternative result, which is better when $k = \omega(\sqrt{n}/\log n)$.

Theorem 9 *The weighted* MIN-DIST *problem can be solved in time $O(n \log^4 n)$ and space $O(n \log n)$.*

Proof: To avoid technicalities, we assume that no two points have the same x-coordinate. Without loss of generality we assume that the points are sorted with respect to their x-coordinates. Thus the input consists in $k > 0$ and $((x_1, y_1, \mu_1), \ldots, (x_n, y_n, \mu_n))$ with $x_1 < \ldots < x_n$. Since the x_i's play no role, we suppose that

the input is reduced to k and $((y_1, \mu_1), \ldots, (y_n, \mu_n))$. Thus the weighted MIN-DIST problem corresponds to MIN-MAX PARTITION(H) with $\Sigma = \mathbb{R} \times \mathbb{R}^+$ and

$$H((z_1, \mu_1), \ldots, (z_r, \mu_r)) = \min_{z \in \mathbb{R}} \max_{1 \leqslant i \leqslant r} \mu_i |z_i - z|.$$

Obviously the cost function H is non-decreasing, that is, it satisfies condition (i) of Theorem 3.

The algorithm given in [13] relies on a preprocessing step requiring $O(n \log n)$ time and $O(n \log n)$ space which allows to perform the queries $(i, j) \mapsto H((y_i, \mu_i), \ldots, (y_j, \mu_j))$ in time $O(\log^4 n)$. So Theorem 3 yields the desired result. □

□

6 Handling outliers

In this section, we consider a generalization of the MIN-DIST problem where at most h points are allowed to be at distance more than ε from P. These points are outliers in the dataset, and we ignore them so as to make our algorithm more robust to noise. More formally, given $P \subseteq \mathbb{R}^2$ and $h, k \in \mathbb{N}$, the OUTLIER problem consists in finding a k-step function f and a subset $P' \subseteq P$ such that $|P'| \leqslant h$ and $\mathrm{d}(P \setminus P', f)$ is minimized.

When $|P| = n$, the obvious dynamic programming approach gives an $O(n^2 h^2)$ time algorithm. As in practice we expect h to be much smaller than n, our goal is to improve the dependency on n.

We first give an algorithm for the decision problem corresponding to OUTLIER, with running time $O(nh^2)$ when the input points are given in sorted order. It is essentially the same algorithm as our decision algorithm for MIN-DIST, except that it needs to consider removing up to h outliers from a combination of points below and above the steps. Using sorted matrix searching and this decision algorithm, we obtain a simple $O(nh^2 \log n)$ time algorithm to solve the OUTLIER problem.

6.1 Decision algorithm

We denote by $p_i = (x_i, y_i)$, $1 \leqslant i \leqslant n$ the points in P. We assume that these points are in sorted order in the sense that $x_i < x_{i+1}$ for all i. (To simplify the exposition, we assume that no two points in P have the same x-coordinate.) We denote $P[i, j] = \{p_i, \ldots, p_j\}$ and $Y[i, j] = \{y_i, \ldots, y_j\}$.

For all integers $i \leqslant j \in [1, n]$ and $q \in [0, h]$, we define $\Delta[i, j, q]$ as the minimum of the diameters of $\{y_i \mid i \in I\}$ over all $I \subseteq [i, j]$ such that $|I| \geqslant j - i + 1 - q$. For all $i \in [1, n]$ and $q \in [0, h]$, we define an integer $\mathrm{next}_\varepsilon(i, q)$ which is the largest value of j such that $\Delta[i, j - 1, q]$ is smaller than ε. In other words, $[i, j - 1]$ is the largest interval starting at i such that there exists a 1-step function with distance at most ε from $P[i, j - 1]$, allowing q outliers in $P[i, j - 1]$.

Suppose that the $q + 1$ smallest y-coordinates (with repetitions) in $P[i, j]$ are $m_0 \leqslant \ldots \leqslant m_q$ and the $q + 1$ largest are $\hat{m}_q \leqslant \ldots \leqslant \hat{m}_0$. Then we have $\Delta[i, j, q] = \min_{a+b=q} \hat{m}_a - m_b$ (of course this statement should be adapted if $j - i < q$). The following observation follows immediately.

Proposition 10 *Given sorted lists of the largest $q + 1$ numbers and the smallest $q + 1$ numbers in $\{y_i, y_{i+1}, \ldots y_j\}$, we can compute $\Delta[i, j, q]$ in $O(q)$ time.*

This observation allows us to compute efficiently all the values of $\mathrm{next}_\varepsilon(i, q)$.

Lemma 11 *Given $\varepsilon > 0$, we can compute all the values of $\mathrm{next}_\varepsilon(i,q)$ for $i \in [1,n]$ and $q \in [0,h]$ in time $O(nh^2)$.*

Proof: We fix a value of q, and we will show that for this value of q, we can compute in $O(nq)$ time all the values of $\mathrm{next}_\varepsilon(i,q)$ for $i \in [1,n]$. The result then follows immediately.

During the course of our algorithm, we use two pointers $1 \leqslant i \leqslant j \leqslant n$ whose initial values are $i = j = 1$. We increment one of them at each step, until we reach $i = n$, and we maintain $\Delta[i,j,q]$ at each step. We first show how to do it in amortized $O(q)$ time per step. For this purpose, we maintain a third pointer m such that $i \leqslant m \leqslant j$ and, together with it, sorted lists $L_{i'}^m$ of the $q+1$ largest values and the $q+1$ smallest values in $Y[i',m]$ for all $i' \in [i,m]$. We also maintain a sorted list L of the $q+1$ largest values and the $q+1$ smallest values in $Y[m,j]$. From these lists, we can easily get $\Delta[i,j,q]$ in $O(q)$ time, by merging the lists L_i^m and L, and by applying Proposition 10. Since i and j are incremented a total of less than $2n$ times, it contributes only $O(nq)$ to the running time. Now we explain how we maintain the list L and the lists L_i^m. Suppose that at the current step, j is incremented. Then we simply update L by inserting y_j, if y_j is among the $q+1$ smallest (or largest) numbers in $Y[m,j]$. If i is incremented and $i < m$, then we simply delete L_i^m. Finally, if i is incremented and becomes equal to m, then we delete all the sorted lists and we set $m = j$. Now we can recompute the lists $L_{i'}^m$ for $i \leqslant i' \leqslant m$ from right to left in amortized time $O(q)$ per list. For a given value of i', this scheme only computes one list $L_{i'}^m$ for one value of m. So over the course of the algorithm, we compute less than $2n$ lists, each one in amortized $O(q)$ time. So overall, maintaining these lists takes $O(nq)$ during the course of the algorithm.

Now we explain how we compute the values of $\mathrm{next}_\varepsilon(\cdot,\cdot)$. We traverse the integer interval $[1,n]$ from left to right using two pointers $i \leqslant j$. Initially, $i = j = 1$ and we increment j until $\Delta[1,j,q] > \varepsilon$. At this point, we set $\mathrm{next}_\varepsilon(1,q) = j$ and we increment i. Now $i = 2$ and, if $\Delta[i,j,q] > \varepsilon$, we set $\mathrm{next}_\varepsilon(i,q) = j$ and repeat the process. Otherwise, we increment j until $\Delta[i,j,q] > \varepsilon$, at which point we set $\mathrm{next}_\varepsilon(i,q) = j$. We repeat this process until $i = j = n$.

It is easy to see that this algorithm is correct. To analyze it, notice that both i and j are incremented at most $n-1$ times, and each time we increment them, we need to spend amortized time $O(q)$. □

□

We can now solve the decision problem in time $O(nh^2)$. First notice that, if the answer to the decision problem is positive, then one of the functions that are solution to this problem is constant only over intervals of the form $[i, \mathrm{next}_\varepsilon(i,q))$. The proof is the same as the proof of Lemma 1. It allows us to speed-up the following dynamic programming approach, and obtain an $O(nh^2)$ time bound. We proceed by decreasing values of i, and we maintain for each index i, and for each $q \in [0,h]$, the minimum number of steps of a step function that has distance at most ε to $P[i,n]$ using q outliers. We spend $O(h^2)$ time for each value of i. So we obtain this result:

Proposition 12 *The decision problem with h outliers and distance ε can be solved in time $O(nh^2)$.*

6.2 Optimization algorithm

We first sort P according to the x-coordinates, which takes $O(n \log n)$ time. Note that the solution to the optimization problem corresponds to a value ε which is half the difference between two values y_i and y_j. So after sorting the set of y-coordinates, we reduce our problem to a search in a sorted matrix with a decision algorithm that runs in time $O(nh^2)$. Hence we obtain:

Theorem 13 *The* OUTLIER *problem can be solved in time $O(nh^2 \log n)$.*

7 Concluding remarks

Our $O(n \log n)$ time algorithm for MIN-DIST is optimal in the sense that we have a matching worst case lower bound of $\Omega(n \log n)$. However, it may be possible to find an algorithm with improved running time depending on k, for instance with time bound $O(n \log k)$. We don't know how to solve this problem, but we can prove an $\Omega(n \log k)$ lower bound on the decision problem as follows. First, it follows directly from Ben-Or's result on algebraic computation trees [5, 8] that the problem of deciding whether n real numbers $\{x_1, \ldots, x_n\}$ are in $\{1, \ldots, k\}$ requires $\Omega(n \log k)$ time. Then there exists a k-step function at distance 0 from the point-set $\{(1,1), \ldots, (k,k), (x_1, x_1), \ldots, (x_n, x_n)\}$ if and only if the x_i's are all in $\{1, \ldots, k\}$; it shows that deciding whether there is a k-step function at distance 0 is harder than the problem of deciding whether n real numbers are in $\{1, \ldots, k\}$, and thus this decision problem has an $\Omega(n \log k)$ lower bound in the algebraic computation tree model.

It would also be interesting to look at the L_1 version: find a k-step function f that minimizes $\sum_{p \in P} \mathrm{d}(p, f)$. This problem is of the min-sum partition type, as opposed to the problems studied in this paper which are of the min-max type. A related min-sum problem can be solved efficiently: given an integer $k > 0$ and a set of weighted points from the real line $P = \{(x_1, w_1), \ldots, (x_n, w_n)\} \subset \mathbb{R}^2$ with $x_1 \leqslant x_2 \leqslant \ldots \leqslant x_n$, we want to find an optimal quantization of P, which is a partition of $\{1, \ldots, n\}$ into k intervals $I_1, \ldots, I_k$ minimizing $\sum_{i=1}^{k} \omega(I_i)$, where

$$\omega(I_i) = \min_{x \in \mathbb{R}} \left\{ \sum_{j \in I_i} w_j (x_j - x)^2 \right\}.$$

As an application of an algorithm for finding a shortest k-link path in a graph with the concave Monge property, Aggarwal et al. [3] obtained an $O(n\sqrt{k \log n} + n \log n)$ time algorithm to compute an optimal quantization. (This result relies on parametric searching.) Unfortunately, the L_1 metric does not have the Monge property. It would be interesting to know if it is possible to do better than the obvious dynamic programming approach in the L_1 case.

References

[1] P. Agarwal, S. Har-Peled, and H. Yu. Robust shape fitting via peeling and grating coresets. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 182–191, 2006.

[2] P. Agarwal and M. Sharir. Efficient algorithms for geometric optimization. *Computing Surveys*, 30, 1998.

[3] A. Aggarwal, B. Schieber, and T. Tokuyama. Finding a minimum-weight k-link path in graphs with the concave Monge property and applications. *Discrete and Computational Geometry*, 12:263–280, 1994.

[4] R. Atanassov, P. Bose, M. Couture, A. Maheshwari, P. Morin, M. Paquette, M. Smid, and S. Wuhrer. Algorithms for optimal outlier removal. *Journal of Discrete Algorithms*, 2(7):239–248, 2009.

[5] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 80–86, 1983.

[6] M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. LATIN 2000: Theoretical Informatics, 4th Latin American Symposium*, pages 88–94, 2000.

[7] C. Buragohain, N. Shrivastava, and S. Suri. Space efficient streaming algorithms for the maximum error histogram. In *Proc. 23rd International Conference on Data Engineering*, pages 1026–1035, 2007.

[8] P. Bürgisser, M. Clausen, and M. Shokrollahi. *Algebraic Complexity Theory*. Springer, 1997.

[9] G. Frederickson. Optimal algorithms for tree partitioning. In *Proc. 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 168–177, 1991.

[10] G. Frederickson and D. Johnson. Generalized selection and ranking: Sorted matrices. *SIAM Journal on Computing*, 13(1):14–30, 1984.

[11] H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th Annual ACM Symposium on Theory of Computing*, pages 135–143, 1984.

[12] M. Goodrich. Efficient piecewise-linear function approximation using the uniform metric. *Discrete and Computational Geometry*, 14(4):445–462, 1995.

[13] S. Guha and K. Shim. A note on linear time algorithms for maximum error histograms. *IEEE Transactions on Knowledge and Data Engineering*, 19(7):993–997, 2007.

[14] S. Har-Peled and Y. Wang. Shape fitting with outliers. *SIAM Journal on Computing*, 33(2):269–285, 2004.

[15] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[16] Y. Ioannidis and V. Poosala. Histogram-based solutions to diverse database estimation problems. *IEEE Data Eng. Bull.*, 18(3):10–18, 1995.

[17] P. Karras, D. Sacharidis, and N. Mamoulis. Exploiting duality in summarization with deterministic guarantees. In *Proc. 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 380–389, 2007.

[18] M. Lopez and Y. Mayster. Weighted rectilinear approximation of points in the plane. In *Proc. LATIN 2008: Theoretical Informatics, 7th Latin American Symposium*, volume 4957 of *Lecture Notes in Computer Science*, pages 642–653, 2008.

[19] Y. Mayster and M. A. Lopez. Approximating a set of points by a step function. *Journal of Visual Comunication and Image Represententation*, 17(6):1178–1189, 2006.

[20] J. Díaz-Báñez and J. Mesa. Fitting rectilinear polygonal curves to a set of points in the plane. *European Journal of Operational Research*, 130(1):214–222, 2001.

[21] D. Wang. A new algorithm for fitting a rectilinear x-monotone curve to a set of points in the plane. *Pattern Recognition Letters*, 23(1-3):329–334, 2002.