ELSEVIER

Computational Geometry 10 (1998) 23–29

Computational Geometry Theory and Applications

Optimal slope selection via cuttings[☆]

Hervé Brönnimann[a,*], Bernard Chazelle[b]

[a] INRIA Sophia-Antipolis, B.P. 93, 2004, Route des Lucioles, 06902 Sophia-Antipolis, France
[b] Department of Computer Science, Princeton University, Princeton, NJ 08544, USA

Communicated by J. Matoušek; submitted 22 January 1996; accepted 26 June 1997

Abstract

We give an optimal deterministic $O(n \log n)$-time algorithm for slope selection. The algorithm borrows from the optimal solution given in (Cole et al., 1989) but avoids the complicated machinery of the AKS sorting network and parametric searching. This is achieved by redesigning and refining the $O(n \log^2 n)$-time algorithm of Chazelle et al. (1993) with the help of additional approximation tools. © 1998 Elsevier Science B.V.

Keywords: Deterministic optimal algorithm; Arrangements; Sorting

1. Optimal slope selection

The problem is to compute the line defined by two of n given points that has the median slope among all $\binom{n}{2}$ such lines. Equivalently, the problem can be stated as that of selecting the median-abscissa vertex of the arrangement $\mathcal{A}(\mathcal{L})$ of a set $\mathcal{L}$ of n lines [6]. For generality, we set out to compute the vertex with rank I^* from left to right, for any given $1 \leqslant I^* \leqslant \binom{n}{2}$. This problem is connected to the problem of sorting $X + Y = \{x + y,\ x \in X,\ y \in Y\}$ [8], and occurs in statistics where the line of median slope is called the Theil–Sen estimator, and is a robust indicator of the slope of the regression line through the points [7].

An optimal deterministic solution was given by Cole et al. in [6], but it requires the use of the AKS sorting network [1] and parametric searching [12], and its analysis is fairly complicated. Simple randomized algorithms were subsequently designed by Dillencourt et al. [7] and Matoušek [11]. An optimal deterministic algorithm was given by Katz and Sharir in [9], building on the simple $O(n \log^2 n)$-time algorithm of Chazelle et al. [4] and using expander graphs plus an approximation tool from [6]. The solution we give here also optimizes the algorithm of [4], using only elementary data structures. The exposition is entirely self-contained, besides the construction of ε-nets [2,10].

[☆] A preliminary version of this paper appeared in the Seventh Canadian Conference on Computational Geometry, Saskatoon, Canada, 1994, pp. 99–103.

[*] Corresponding author.

PII S0925-7721(97)00025-4

2. Definitions and notation

The lines in $\mathcal{L}$ are numbered in order of decreasing slopes. To ease the exposition, we suppose that the lines are algebraically independent, which in particular implies that vertices of their arrangement are incident upon only two lines and have distinct abscissae. This assumption can be removed easily by a more precise treatment of degeneracies. (Indeed, it suffices to treat a vertex $v = l \cap l'$ as a triplet $(v, N(l), N(l'))$, where line l is numbered $N(l)$. Lexicographic order disambiguates between vertices having the same abscissa.) Any vertical line $x = x_0$ cuts $\mathcal{A}(\mathcal{L})$ in n points (whose ordinates are called the *intercepts*), and the number of vertices on it or to its left is noted $v(\mathcal{L}, x_0)$. More generally, we let $v(\mathcal{L}, S)$ denote the number of vertices of $\mathcal{A}(\mathcal{L})$ inside *any* region S. The vertical ordering of the lines at $x = x_0$ defines a permutation π_{x_0} of the lines, with $\pi_{-\infty}$ being the identity. (If $x = x_0$ passes through a vertex, we make the convention that the two meeting lines are in the same order as at $x = x_0 + \varepsilon$, for infinitesimally small $\varepsilon > 0$.) The number of inversions of this permutation is denoted $I(\pi_{x_0})$ and is exactly $v(\mathcal{L}, x_0)$.

For our purposes, an *ε-net* for a set $\mathcal{L}$ of lines is a subset N such that every segment intersecting more than $\varepsilon|\mathcal{L}|$ of the lines of $\mathcal{L}$ intersects at least one line of N. Matoušek [10] gave an algorithm that computes ε-nets in linear time, if ε is a constant. (See also [2,5].) The only subroutine needed by this algorithm is one that computes the arrangement of a constant number of lines of $\mathcal{L}$.

Borrowing terminology from [6], let us partition a permutation π into a collection $\mathcal{B}$ of at most $2n/m$ blocks, each containing at most m lines consecutive in π; we obtain what is called an *m-blocked permutation* $\pi_\mathcal{B}$. If any two lines belonging to different blocks are ordered as in π_{x_0}, and all inversions within a block of $\pi_\mathcal{B}$ actually occur in π_{x_0}, we say that $\pi_\mathcal{B}$ is *left-compatible* with π_{x_0}. Note that π_{x_0} might contain inversions absent from $\pi_\mathcal{B}$. Because at most $\binom{m}{2}$ inversions can occur within a block of size m, we have

$$I(\pi_\mathcal{B}) \leqslant I(\pi_{x_0}) \leqslant I(\pi_\mathcal{B}) + nm. \tag{1}$$

Reversing the order along the x-axis gives the concept of *right-compatibility*, and we have the converse

$$I(\pi_\mathcal{B}) - nm \leqslant I(\pi_{x_0}) \leqslant I(\pi_\mathcal{B}). \tag{2}$$

Therefore, we see that maintaining a blocked partition compatible with a permutation gives a good estimate on the number of inversions of this permutation: the smaller the blocks, the finer the estimate. The rank of a vertex (x, y) is $\operatorname{rank}(x) = I(\pi_x)$, and the problem is to find the vertex v^* of $\mathcal{A}(\mathcal{L})$ of abscissa x^* with $\operatorname{rank}(x^*) = I^*$.

3. Updating blocked permutations

Assume we have an m-blocked permutation π left- (respectively right-) compatible with π_x for some x, and we wish to have an m-blocked permutation π' compatible with $\pi_{x'}$ for a given $x' > x$ (respectively $x' < x$) such that $|\operatorname{rank}(x') - \operatorname{rank}(x)| = \mathrm{O}(nm)$. We modify and simplify a technique called *reblocking* that was used in [6]. Let us process the lines of $\mathcal{L}$ in the order given by π. Before processing line l, assume we have a linked list of stacks, $s_1, \dots, s_q$, each of them containing between $\lceil m/2 \rceil$ and m elements. Initially, only s_1 is in the list, and it is filled with the first m elements of π. For each stack s_i, we keep two counters: the number $\operatorname{size}(s_i)$ of elements, and the lowest intercept at

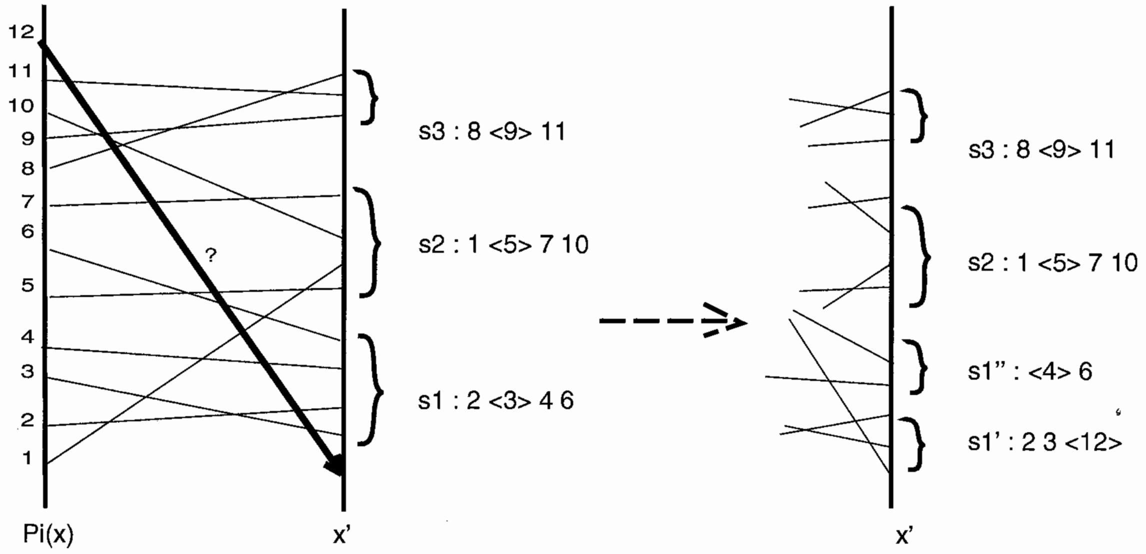

Fig. 1. The lines numbered 1–11 are 4-blocked in the order of π_x, and reblocked at x', with $I'' = 7$ more intersections detected. The stacks are described by the lines in order, with $\mathrm{low}(s_i)$ between brackets. Upon inserting 12 into the list of stacks, we step down the list to s_1, adding 7 more intersections. Then s_1 is split into s_1' and s_1'', adding 2 more intersections.

x' of a line of s_i, which we denote $\mathrm{low}(s_i)$. We also keep a global counter I'', initially set to 0. This counter will serve to count the additional inversions between π and π'.

To insert a line l into our linked stacks, we first try to insert it into s_q, the last stack in the list. If the intercept $y(l)$ of l at x' is lower than the lowest intercept of s_q, we try to insert l into s_{q-1}, and iterate if necessary. Each time we go down the list from s_i to s_{i-1}, we increase I'' by $\mathrm{size}(s_i)$. Should we reach s_1, we insert l there and update $\mathrm{low}(s_1)$.

If after inserting a line into a stack s_i, this stack has $m+1$ elements, we need to split it. To do so, we compute the median y of the intercepts $(y(l))_{l\in s_i}$ at x' of the lines in this stack, and reinsert the lines into a new stack s_i' (respectively s_i'') according to whether their intercept is smaller than or equal to (respectively larger than) y. The lines are inserted in the same order as which they have been put into s_i. Each time a line is inserted into s_i', I'' is incremented by the current $\mathrm{size}(s_i'')$. Then blocks $s_{i+1},\dots,s_q$ are renumbered $s_{i+2},\dots,s_{q+1}$, and s_i' (respectively s_i'') becomes the new s_i (respectively s_{i+1}) if necessary.

When the whole round of insertions finishes, we have a new m-blocked permutation π' obtained by concatenating all the stacks in order, which by construction is compatible with $\pi_{x'}$. If I is the number of inversions of π, we claim that the number of inversions of π' is $I' = I + I''$. Note that we insert the lines in the order given by π. Inside the blocks of π, the lines need not be in the actual order corresponding to that abscissa; however, all the inversions of π have been accounted for in I (including inversions between two lines of the same block). To start with, we observe that we count an inversion between l' and l'' in I'' only if $y(l') < y(l'')$ at x (which implies that l' is before l'' in π), and if l' and l'' are assigned to different blocks in π' before or after splitting a stack. In this case, the intersection is being witnessed: $y(l') \geqslant \mathrm{low}(s) > y(l'')$ for some stack s. Once two elements have been inverted, they will remain in the same order in any subsequent reblocking, for this would imply $y(l') < y(l'')$ at abscissae greater than x', meaning that l and l' have two intersection points, which is impossible. If two lines are in the same block in π', however, it follows from the construction that they are in the same order as in π. Since we cannot count an intersection twice, I'' corresponds exactly to the number of additional inversions from π to π', which establishes our claim.

The subtle point is that the computation of π' is done in $O(n)$ time. This can be most easily seen by the fact that the total insertion time is proportional to the number of times an element steps down in the list. But doing so adds at least $m/2$ inversions, and we know that there are at most $O(nm)$ inversions between π and π'. Therefore going down the list cannot happen more than $O(n)$ times. The time taken by a split is also $O(m)$, and the number of split operations is bounded by the final number of blocks in π', which is $O(n/m)$. Thus we have Lemma 1.

Lemma 1 (Reblocking). *Given an m-blocked permutation π left- (respectively right-) compatible with π_x for some x, it is possible to compute in $O(n)$ time an m-blocked permutation π' left- (respectively right-) compatible with $\pi_{x'}$ for any given $x' > x$ (respectively $x' < x$) such that $|\text{rank}(x') - \text{rank}(x)| = O(nm)$.*

The same reblocking strategy also works for halving the size of the blocks of π. Simply halve the size of all the blocks of π as we did in splitting a stack in the paragraph above.

Lemma 2 (Halving). *Given an m-blocked permutation π left- (respectively right-) compatible with π_x for some x, it is possible to compute in $O(n)$ time an $(m/2)$-blocked partition which is still left- (respectively right-) compatible with π_x.*

4. The algorithm

We first define a *vertical slab* (l, r) as the portion $\{(x, y)\colon\ l < x \leqslant r\}$. The algorithm maintains an m_l-blocked permutation $\pi(l)$ which is left-compatible with π_l, and an m_r-blocked permutation $\pi(r)$ which is right-compatible with π_r. It also maintains the number of inversions I_l (respectively I_r) of $\pi(l)$ (respectively $\pi(r)$). Finally, it stores a collection $\mathcal{T}$ of vertical trapezoids covering (l, r), along with their *conflict lists* (the set of lines crossing them). By analogy with [3,4], we call $\mathcal{T}$ a *cutting* for the slab (l, r). For accounting purposes, we give a size of 1 to an empty conflict list. In this way, the total size of the conflict lists becomes an accurate indicator of the size of the data structure. The algorithm proceeds in a logarithmic number of *steps*, and maintains the four following invariants at step j (for some large enough constant c).

(I1) $I_l + 2nm_l \leqslant I^* \leqslant I_r - 2nm_r$.
(I2) $I_r - 4nm_r < I^* < I_l + 4nm_l$.
(I3) Any trapezoid of $\mathcal{T}$ is crossed by at most n/c^j lines.
(I4) The total size of the conflict lists of $\mathcal{T}$ is at most cn.

Informally, (I1) says that the vertex sought lies comfortably within the slab (l, r), because of Eqs. (1) and (2); (I2) means that the blocked permutations are no finer than needed; (I3) guarantees that the number of steps will be logarithmic; and (I4) guarantees that the work in a single step is $O(n)$.

A step of the algorithm can be a *slab refinement step*, whose purpose is to narrow the slab (l, r), or a *cutting refinement step*, which subdivides the trapezoids of $\mathcal{T}$. Before explaining how they will be interleaved in the algorithm, let us describe how to perform them.

We halve the slab (l, r) in the same fashion as [4]. A point is said to be *critical* if (i) it lies in the interior of (l, r), and either (ii) it is at the intersection of some line and the upper (lower) boundary of a trapezoid or (iii) it is a vertex of a trapezoid. Intersections along l and r are not critical points. For accounting purposes, we include the vertices of (iii) with a multiplicity equal to the number of lines crossing the incident vertical boundaries. Let V denote the multiset of all the critical points. From (I4), we know that V has size $\mathrm{O}(n)$. We compute its median abscissa h.

We choose a *winning slab* between (l, h) and (h, r) in the following fashion: for each side s (either l or r), we reblock $\pi(s)$ at $x = h$. We then keep halving m_s, until (I1), (I2) are restored. Reblocking and halving are described in the previous section and each takes $\mathrm{O}(n)$ time. Potentially, this could be a very long process (in particular if x^* is very near h). To avoid this, we organize the computation as follows: each side is run in parallel (think of it as having two distinct processors, or as giving the odd cycles to side l and the even cycles to side r), and the computation stops when (I1), (I2) are restored for either side s (but not necessarily both). If s equals l, the winning slab is (h, r), otherwise it is (l, h). We call this the *slab selection process*.

Once we know the winning slab w, we discard trapezoids of $\mathcal{T}$ that don't intersect w, keep only the portion inside w of those which intersect the line $x = h$ and update their conflict list, and keep those entirely contained in $\mathcal{T}$ as they are. This yields a different cutting $\mathcal{T}'$. If (l, h) is the winning slab, we leave $\pi(l)$ unchanged, and let h replace r, with $\pi(h)$ being the permutation reblocked from $\pi(r)$ at $x = h$ and halved as many times as in the slab selection process. We proceed symmetrically if (h, r) is the winning slab. Observe that (I1)–(I4) are maintained, and that the number of critical points in the winning slab has been at least halved compared to those in the slab (l, r). This concludes the description of the slab refinement step.

If we kept iterating on this process, we would quickly run out of critical points. So, we repeat the slab refinement step a constant number of times until the number of all critical points drops below $n/(c \log c)^2$, at which point we compute a $(1/(4c))$-net of size $\mathrm{O}(c \log c)$ for each of the conflict lists [2,5,10], and compute its vertical trapezoidal map inside the relevant trapezoid. Finally, we increment j by one. Any of the four sides of a new trapezoid is intersected by at most $(1/4c)(n/c^j)$ lines, hence the new cutting satisfies (I3) for the new value $j' = j + 1$ of j. Note that the new multiset of critical points is of size at most $c_1 n$ (for a constant c_1 not depending on c but on the size of the net). Following [4], we say that an edge of a trapezoid is *free* if it runs entirely across the slab. We remove any free edge if doing so does not create trapezoids violating (I3). Since the free edges are vertically ordered, after removal, there can be at most $c_2 c^j$ such edges, accounting for at most $2c_2 n$ of the conflict lists elements (for another constant c_2 independent of c). In order to maintain (I4), we observe that each element in the conflict list can either be charged to a new critical point (in number $c_1 n$), to an intersection with the two vertical bounding lines (exactly $2n$ of them), or to a free edge (in total number $c_2 n$). Therefore, taking $c \geqslant c_1 + 2c_2 + 2$ ensures that the new cutting satisfies (I4) as well. This concludes the description of the cutting refinement step.

We organize the sequence of steps as follows: we refine the slab until at least one refinement for the cutting is needed. This can end in one of two ways: either we find that $x^* = h$ because we refined the blocked permutation $\pi(h)$ until it becomes the permutation π_h and $I_h = I^*$; or n/c^j becomes less than 1 after refining the cutting. In the latter case, the full arrangement between l and r is available,

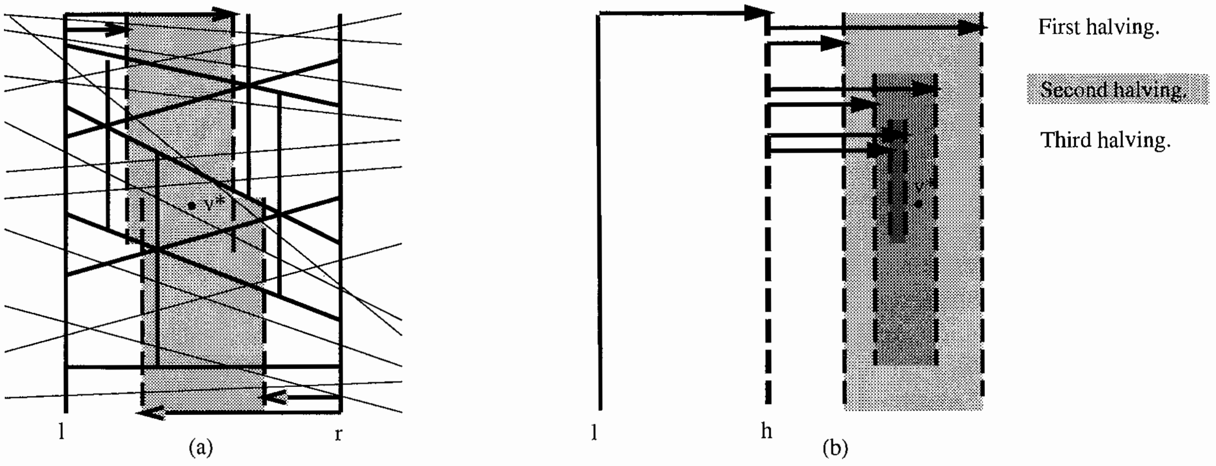

Fig. 2. (a) The cutting and blocked permutations for $\mathcal{L}$ at $x = l$ and $x = r$. No more than four lines cross a trapezoid. Shadowed are the loci of vertices of rank between $I_l + 2nm_l$ and $I_l + 4nm_l$ (respectively between $I_r - 4nm_r$ and $I_r - 2nm_r$). Invariants (I1), (I2) guarantee that v^* lies in the shadows. (b) During a slab refinement step, we reblock $\pi(l)$ at $x = h$. Displayed is the evolution of the shadows as we repeatedly halve until (by the third halving) v^* leaves the shadow, so we keep the permutation after the second halving as the final blocked permutation.

and its vertices are in number less than cn. We compute rank(l) exactly in $O(n \log n)$ time, and select the vertex of rank $I^* - \text{rank}(l)$ in that list. In both cases, the algorithm succeeds in finding the vertex of rank I^*.

5. Running time analysis

As we observed above, the number of critical points is at least halved during a slab refinement step. Therefore, no more than a constant number of slab refinement steps can occur between two consecutive cutting refinement steps. The number of cutting refinement steps is $O(\log n)$, since at each such step the maximum size of the conflict lists decreases by a factor of c. Thus the total number of refinement steps is $O(\log n)$.

But this number is not necessarily a good indicator of the running time: because we halve repeatedly in the slab selection process, a slab refinement step could take substantially more than $O(n)$ time. However, we can show that this is not the case in the amortized sense: let h_i be the median of the critical points at slab refinement step i, and let $K_i = |\text{rank}(h_i) - I^*|$; let $0 \leqslant l_0 \leqslant \cdots \leqslant l_k$, $k = O(\log n)$, be the subsequence of such steps for which the winning slab is (l, h_i), and let similarly $0 \leqslant r_0 \leqslant \cdots \leqslant r_{k'}$, $k' = O(\log n)$, be the subsequence of such steps for which (h_i, r) is the winning slab.

At the beginning of step l_j, $j < k$, we denote the block size of $\pi(r)$ by $m_{r,j}$, and after the halvings the block size of $\pi(h_j)$ becomes $m_{r,j+1}$. Thus the number of halvings during step l_j is $k_{l_j} = \log(m_{r,j}/m_{r,j+1})$. From (I1), (I2) and Eq. (2), we obtain $k_{l_j} = \log(K_{l_{j-1}}/K_{l_j}) + O(1)$. Thus, the total number of halvings performed on $\pi(r)$ throughout the entire algorithm is

$$\sum_{0 \leqslant j < k} k_{l_j} = \sum_{0 \leqslant j < k} (\log K_{l_j} - \log K_{l_{j+1}}) + O(k) = \log K_{l_0} - \log K_{l_k} + O(k) = O(\log n),$$

since $K_i \leqslant \binom{n}{2}$ for any step i. Similarly, $\sum_{0\leqslant j<k'} k_{r_j} = \mathrm{O}(\log n)$. Therefore the total number of halvings (on either side) during all slab refinement steps is also $\mathrm{O}(\log n)$.

Since each reblocking, halving, and cutting refinement step takes $\mathrm{O}(n)$ time, the total running time of the algorithm is $\mathrm{O}(n \log n)$ as claimed, and the storage is $\mathrm{O}(n)$.

Acknowledgements

We wish to thank Bill Steiger, Jirka Matoušek and two anonymous referees for helpful discussions and suggestions to improve the presentation of this paper.

References

[1] M. Ajtai, J. Komlós, E. Szemerédi, Sorting in $c \log n$ parallel steps, Combinatorica 3 (1983) 1–19.

[2] H. Brönnimann, B. Chazelle, J. Matoušek, Product range spaces, sensitive sampling, and derandomization, in: Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci., 1993, pp. 400–409; to appear in SIAM J. Comput.

[3] B. Chazelle, Cutting hyperplanes for divide-and-conquer, Discrete Comput. Geom. 9 (2) (1993) 145–158.

[4] B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir, Diameter, width, closest line pair and parametric searching, Discrete Comput. Geom. 10 (1993) 183–196.

[5] B. Chazelle, J. Matoušek, On linear-time deterministic algorithms for optimization problems in fixed dimension, in: Proc. 4th ACM–SIAM Sympos. Discrete Algorithms, 1993, pp. 281–290.

[6] R. Cole, J. Salowe, W. Steiger, E. Szemerédi, An optimal-time algorithm for slope selection, SIAM J. Comput. 18 (1989) 792–810.

[7] M.B. Dillencourt, D.M. Mount, N.S. Netanyahu, A randomized algorithm for slope selection, Internat. J. Comput. Geom. Appl. 2 (1992) 1–27.

[8] L.H. Harper, T.H. Payne, J.E. Savage, E. Strauss, Sorting $X + Y$ in $\mathrm{O}(n^2)$ comparisons, Comm. ACM 18 (1975) 347–349.

[9] M.J. Katz, M. Sharir, Optimal slope selection via expanders, Inform. Process. Lett. 47 (1993) 115–122.

[10] J. Matoušek, Approximations and optimal geometric divide-and-conquer, in: Proc. 23rd Annu. ACM Sympos. Theory Comput., 1991, pp. 505–511; also to appear in J. Comput. Syst. Sci.

[11] J. Matoušek, Randomized optimal algorithm for slope selection, Inform. Process. Lett. 39 (1991) 183–187.

[12] N. Megiddo, Applying parallel computation algorithms in the design of serial algorithms, J. ACM 30 (1983) 852–865.