Finding Axis-Parallel Rectangles of Fixed Perimeter or Area Containing the Largest Number of Points[*]

Haim Kaplan[1], Sasanka Roy[2], and Micha Sharir[3]

1 **Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
`haimk@tau.ac.il`
2 **Indian Statistical Institute, Kolkata, India**
`sasanka.ro@gmail.com`
3 **Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
`michas@tau.ac.il`

Abstract

Let P be a set of n points in the plane in general position, and consider the problem of finding an axis-parallel rectangle with a given perimeter, or area, or diagonal, that encloses the maximum number of points of P. We present an exact algorithm that finds such a rectangle in $O(n^{5/2}\log n)$ time, and, for the case of a fixed perimeter or diagonal, we also obtain (i) an improved exact algorithm that runs in $O(nk^{3/2}\log k)$ time, and (ii) an approximation algorithm that finds, in $O\left(n+\frac{n}{k\varepsilon^5}\log^{5/2}\frac{n}{k}\log\left(\frac{1}{\varepsilon}\log\frac{n}{k}\right)\right)$ time, a rectangle of the given perimeter or diagonal that contains at least $(1-\varepsilon)k$ points of P, where k is the optimum value.

We then show how to turn this algorithm into one that finds, for a given k, an axis-parallel rectangle of smallest perimeter (or area, or diagonal) that contains k points of P. We obtain the first subcubic algorithms for these problems, significantly improving the current state of the art.

1998 ACM Subject Classification F.2.2 Geometrical problems and computations, E.1 Data structures

Keywords and phrases Computational geometry, geometric optimization, rectangles, perimeter, area

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.52

1 Introduction

In the basic problem studied in this paper, we are given a set P of n points in the plane in general position, and a fixed parameter $\tau > 0$, and we seek an axis-parallel rectangle of perimeter 2τ that encloses the maximum number of points of P. We denote this problem as MAX-PTS(τ). We also consider variants of the problem, involving rectangles with other fixed parameters, such as the area or the length of the diagonal. Such problems have been studied

[*] Work by Haim Kaplan has been supported by Grant 1161/2011 from the German-Israeli Science Foundation, by Grant 1841-14 from the Israel Science Foundation, and by the Israeli Centers for Research Excellence (I-CORE) program (center no. 4/11). Work by Micha Sharir has been supported by Grant 2012/229 from the U.S.-Israel Binational Science Foundation, by Grant 892/13 from the Israel Science Foundation, by the Israeli Centers for Research Excellence (I-CORE) program (center no. 4/11), by the Blavatnik Research Fund in Computer Science at Tel Aviv University, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

by many researchers (as we survey below) and arise in statistical clustering and pattern recognition (see [1] and the references therein).

Earlier works on these problems have mainly studied the "dual" version, where we specify k and seek an axis-parallel rectangle of minimum perimeter (or area, or diagonal) that encloses k points of P. This so-called MIN-PERIM(k) (or MIN-AREA(k), or MIN-DIAG(k)) problem has been studied in Aggarwal *et al.* [1], who gave an $O(nk^2 \log n)$-time solution. Eppstein and Erickson [7] gave an algorithm that runs in $O(n \log n + nk^2)$ time and requires $O(n \log n + nk)$ storage. They observed that the k points in the optimal rectangle are among the $O(k)$ rectilinear nearest neighbors of each other. This allowed them to partition the problem into $O(n/k)$ subproblems, each on a subset of $O(k)$ points, and to apply a naive $O(k^3)$-algorithm of Aggarwal *et al.* [1] to each subset. The partition is obtained using a data structure for planar rectilinear nearest neighbors.

Datta *et al.* [5] suggested a different scheme to break the problem into $O(n/k)$ subproblems, each of size $O(k)$. Their algorithm runs within the same time bound as the algorithm of Eppstein and Erickson but requires only linear storage.

Segal and Kedem [15] gave algorithms for MIN-PERIM(k) and MIN-AREA(k), that are linear for very large values of k. Their algorithms run in $O(n + k(n-k)^2)$ time.

A very recent work on this problem, by De Berg *et al.* [6], develops a near linear algorithm for MIN-AREA(k), for small values of k. Their algorithm runs in $O(nk^2 \log n + n \log^2 n)$ time. They also give a $(1-\varepsilon)$-approximation algorithm for the dual MIN-PTS(A) problem, which asks for an axis-parallel rectangle of a given area A that contains the largest number of points of P. Notice that all algorithms for MIN-PERIM(k) and MIN-AREA(k) are cubic in the worst-case for some values of k.

As noted in [6], the variants of the problem involving area are harder. For example, it is no longer the case that the k points in an optimal rectangle are among the $O(k)$ rectilinear nearest neighbors of each other. Our paper too, which handles all three variants, derives faster algorithms for the cases of perimeter or diagonal, and the approximation algorithms that we obtain apply only for these two cases.

Let us return to the case of perimeter. Using an algorithm for MIN-PERIM(k), one can solve the original problem, that we have denoted as MAX-PTS(τ), using binary search in a straightforward manner. The overall cost of this algorithm is $O(\log n)$ times the cost of MIN-PERIM(k).

The converse direction, to turn a given algorithm for MAX-PTS(τ) into an efficient solution of MIN-PERIM(k), is somewhat more involved, but is doable. Indeed, in this paper we first solve MAX-PTS(τ) directly, and then show how to solve MIN-PERIM(k) by a logarithmic number of calls to MAX-PTS(τ).

1.1 Our results

We first present, in Section 2, an algorithm for MAX-PTS(τ) that runs in $O(n^{5/2} \log n)$ time. The method is sufficiently general, so the algorithm also solves the variants where the area or the diagonal of the rectangle are fixed (and we want to maximize the number of points of P that it contains), within the same running time bound.

We then use, in Section 2.3, a simple grid-based construction that allows us to solve MAX-PTS(τ) in an output-sensitive manner. Specifically, the running time improves to $O(nk^{3/2} \log k)$, where k is the output size, the maximum number of points of P contained in such a rectangle. A simple modification of the same approach yields the same improvement for the case of fixed diagonal, but, unfortunately, not for the case of fixed area.

We also obtain, in Section 3, an approximation algorithm that finds, in near-linear time (see Theorem 3 for the precise bound), an axis-parallel rectangle of the given perimeter that contains at least $(1-\varepsilon)k$ points of P, for a prespecified error parameter ε. The approximation algorithm extends to the case of diagonal but not to the case of area.

Finally, we consider the dual problem MIN-PERIM(k), as defined above, and present an algorithm that solves it in $O(nk^{3/2}\log k \log n)$ time, which is only $O(\log n)$ times slower than the running time of our algorithm for MAX-PTS(τ). We obtain this result by reducing MIN-PERIM(k) to a logarithmic number of calls to MAX-PTS(τ). This bound improves (for almost all values of k) the previous best bound $O(n\log n + nk^2)$ of [5, 7]. Our reduction is fairly general, and can also be applied to the cases of area or diagonal. For the case of area, our algorithm is not output-sensitive, and runs in $O(n^{5/2}\log^2 n)$ time. These are the first subcubic algorithms for these problems (for any value of k), improving upon [1, 6, 5, 7, 15]. For the case of area, the algorithm in [6] is faster when k is small; for the cases of perimeter or diagonal, our algorithms, as already noted, are significantly faster for almost all values of k.

2 An exact algorithm for max-pts(τ)

We recall our basic problem: Let P be a set of n points in the plane in general position (in particular, no two points of P have the same x- or y-coordinate), and let τ be a given positive real number. We want to find an axis-parallel rectangle R of perimeter 2τ that contains the largest number of points of P.

Let $\mathcal{Q} = \mathcal{Q}(\tau)$ denote the collection of all axis-parallel rectangles R of perimeter 2τ. Each rectangle $R \in \mathcal{Q}$ can be parameterized by three parameters (x, y, z), where (x, y) is the bottom-left vertex of R, and z is its width (x-span); its height (y-span) is then $\tau - z$. In other words, we identify the rectangles of $\mathcal{Q}$ with the points of $\mathbb{R}^2 \times [0, \tau]$.

A point $p = (p_1, p_2)$ lies in a rectangle $R \in \mathcal{Q}$, parameterized by (x, y, z), if and only if

$$x \le p_1 \le x + z \qquad \text{and} \qquad y \le p_2 \le y + \tau - z. \tag{1}$$

For each $p = (p_1, p_2) \in P$, let K_p denote the set of all rectangles in $\mathcal{Q}$ that contain p. In the parametric 3-space, K_p is a tetrahedron, bounded by the four halfspaces specified in (1). Our problem is now reduced to that of finding a point of maximum *depth* in the arrangement of these n isothetic tetrahedra (i.e., a point contained in the largest possible number of tetrahedra; note that these tetrahedra are indeed translates of one another).

The cross-section of a tetrahedron K_p, at any fixed z, is an axis-parallel rectangle $K_p(z)$, given by

$$p_1 - z \le x \le p_1 \qquad \text{and} \qquad p_2 - \tau + z \le y \le p_2. \tag{2}$$

All the rectangles $K_p(z)$, for $p \in P$, are translates of one another; they are in fact the sets $p - R_0(z)$, for $p \in P$, where $R_0(z) = [0, z] \times [0, \tau - z]$. Let $\mathcal{R}(z)$ denote the collection of these rectangles, for any $z \in [0, \tau]$, and let $\mathcal{A}(z)$ denote the planar arrangement of the rectangles of $\mathcal{R}(z)$. Our problem now is to find a z at which the maximum depth $\Delta(z)$ of a point in $\mathcal{A}(z)$ is maximized.

As z varies from 0 to τ, the rectangles of $\mathcal{R}(z)$ simultaneously deform, as their common width (x-span) increases and their common height (y-span) decreases. The arrangement $\mathcal{A}(z)$ varies continuously, but its combinatorial structure remains unchanged as long as both the left-to-right order of the y-vertical edges of the rectangles, and the bottom-to-top order of the x-horizontal edges of the rectangles, remain unchanged. Under the general position assumption, no pair of left edges, of right edges, of top edges, or of bottom edges, can ever attain the same x- or y-coordinate. Hence, the critical values of z at which the combinatorial structure of $\mathcal{A}(z)$ changes are those at which either the lines supporting the left side of one rectangle and the right side of another coincide, or the lines supporting the bottom side of one rectangle and the top side of another coincide. The set of critical values is therefore

$$\Big\{p_1 - q_1,\ p_2 - q_2 + \tau \mid p = (p_1, p_2) \neq q = (q_1, q_2) \in P\Big\} \cap (0, \tau).$$

There are at most $n(n-1)$ such critical values (for each pair p, q, at most one of $p_1 - q_1$, $q_1 - p_1$ can lie in $(0, \tau)$, and the same holds for $p_2 - q_2 + \tau$, $q_2 - p_2 + \tau$), and we may assume them to be all distinct, by our general position assumption. At each such critical value, either the left edge of one rectangle and the right edge of another rectangle in $\mathcal{R}(z)$, that are adjacent in the left-to-right order of the vertical edges, are swapped in this order, or the bottom edge of one rectangle and the top edge of another rectangle, adjacent in the bottom-to-top order of the horizontal edges, are swapped.

In what follows, we present a data structure that maintains the arrangement $\mathcal{A}(z)$, updates it at each critical value of z, and keeps track of the maximum depth in $\mathcal{A}(z)$ after each update. The data structure requires $O\left(n^{3/2} \log n\right)$ storage, can be initialized in $O\left(n^{3/2} \log n\right)$ time, and each update takes $O\left(n^{1/2} \log n\right)$ amortized time. Using this structure, we obtain our exact algorithm.

▶ **Theorem 1.** *Given a set P of n points in the plane in general position, and a parameter $\tau > 0$, one can find, in $O\left(n^{5/2} \log n\right)$ time, an axis-parallel rectangle of perimeter 2τ that contains the maximum number of points of P. The algorithm requires $O\left(n^{3/2} \log n\right)$ storage.*

Other fixed parameters. As already noted, the approach described so far can also handle, with minor modifications, other fixed parameters, such as the area, or the length of the diagonal, or any parameter that depends on the lengths of the edges of the rectangles, so that the length of one edge uniquely determines the length of the other edge (all the parameters mentioned so far have this property). For example, in the case where our rectangles have a fixed area A, we have to replace (1) and (2) by

$$x \le p_1 \le x+z, \qquad y \le p_2 \le y+A/z, \qquad \text{and} \qquad p_1-z \le x \le p_1, \qquad p_2-A/z \le y \le p_2,$$

respectively. The latter pair of inequalities define K_p, so it is no longer a simplex. Nevertheless, the cross sections $K_p(z)$, for any $z > 0$, are all isothetic rectangles, and the critical values of z are similarly constructed – they are now of the form $|p_1 - q_1|$ or $A/|p_2 - q_2|$. This allows the rest of the analysis to proceed more or less verbatim. The case of fixed diagonal is also handled in a fully analogous manner. That is, these variants can also be solved in $O(n^{5/2} \log n)$ time.

2.1 The data structure

For any fixed value of z, one can compute the maximum depth of $\mathcal{A}(z)$ in $O(n \log n)$ time and $O(n)$ space [12, 14]. If we allow $O(n \log n)$ space (and $O(n \log n)$ time) then we can perform this computation in a straightforward manner by sweeping $\mathcal{R}(z)$ from left to right by a vertical line, while maintaining the cross sections of the rectangles in $\mathcal{R}(z)$ with the sweepline in a dynamic segment tree T [3]. To efficiently recompute the maximum depth after each update of T, we also store at each node of T the maximum depth of a leaf in its subtree. (The *depth* of a leaf is the number of rectangles containing the vertical interval

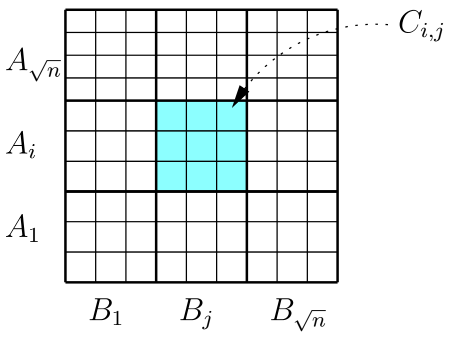

Figure 1 The grid $G(z)$, the coarser grid, its horizontal and vertical slabs, and a single highlighted cell.

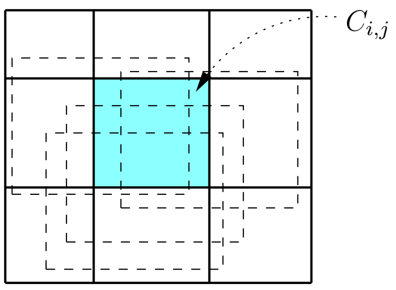

Figure 2 The sets $H_{i,j}$ and $V_{i,j}$, and the respective sets $\mathcal{R}^H_{i,j}$, $\mathcal{R}^V_{i,j}$ of the rectangles forming them, within a single highlighted cell $C_{i,j}$.

that it represents within the sweepline.) An update affects these counters only at the nodes on the two paths to the endpoints of the inserted or deleted segment, and it is therefore straightforward to update these counters when we insert or delete a segment to/from the segment tree, and to propagate them to the root, whose counter represents maximum the depth that we seek.

One could attempt at making this (static) structure dynamic, under swaps of vertical or horizontal rectangle edges, by maintaining all the versions of T, constructed during the sweep, in some persistent data structure, and by updating that structure at each swap of edges. Unfortunately, it is not clear how to perform such updates efficiently. (This is because a swap of two horizontal edges might affect arbitrarily many versions of T; vertical swaps, in contrast, affect only two consecutive versions.) Instead, we present a slower, more symmetric data structure for computing the maximum depth of $\mathcal{A}(z)$, which can be made dynamic at a reasonably low cost.

The grid. Consider the grid $G = G(z)$ formed by the horizontal and vertical lines supporting the edges of the rectangles in $\mathcal{R} = \mathcal{R}(z)$. (The other notations introduced below also depend on z, but we make this dependence implicit from now on, to simplify the notation.) Assuming that z is not critical, G is formed by $2n$ vertical lines and by $2n$ horizontal lines.

Partition G into $\sqrt{n} \times \sqrt{n}$ cells, by the $\sqrt{n} - 1$ horizontal lines and the $\sqrt{n} - 1$ vertical lines whose indices are multiples of $2\sqrt{n}$, referred to as *dividers*; the number of cells in the resulting coarser grid is n. (We ignore in what follows the insignificant rounding issues.) The horizontal dividers partition the plane into $\sqrt{n}$ horizontal slabs, denoted $A_1, \ldots, A_{\sqrt{n}}$ in this bottom-to-top order, and the vertical dividers partition the plane into $\sqrt{n}$ vertical slabs, denoted $B_1, \ldots, B_{\sqrt{n}}$ in this left-to-right order. See Figure 1. We regard each vertical (resp., horizontal) slab as closed at its left (resp., bottom) boundary, and open at its right (resp., top) boundary, so that each vertical (resp., horizontal) line is contained in exactly one vertical (resp., horizontal) slab. Accordingly, each cell in the coarser grid is closed at its left and bottom boundaries, and open at its right and top boundaries, making the cells pairwise disjoint and covering all rectangles.

For each cell $C_{i,j} = A_i \cap B_j$ of the coarser grid, let $v_{i,j}$ denote the number of corners of the rectangles in $\mathcal{R}$ that lie in $C_{i,j}$. We have $\sum_{i,j} v_{i,j} = 4n$, so, on average, each cell contains four vertices, but some cells might contain many more vertices. Still, by construction, no cell contains more than $4\sqrt{n}$ vertices. In fact, we have the stronger property that no vertical or horizontal slab contains more than $4\sqrt{n}$ vertices (At most *two* on each vertical line in a vertical slab, and similarly for horizontal slabs).

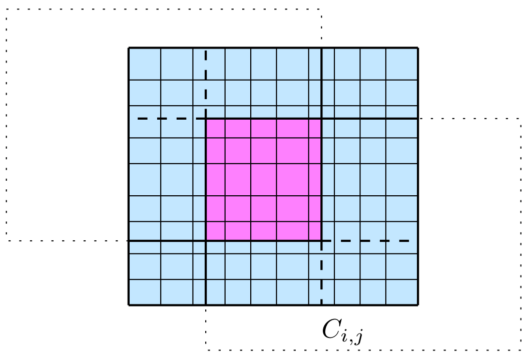

Figure 3 Two short rectangles in $C_{i,j}$, and the partition into query subcells that they induce.

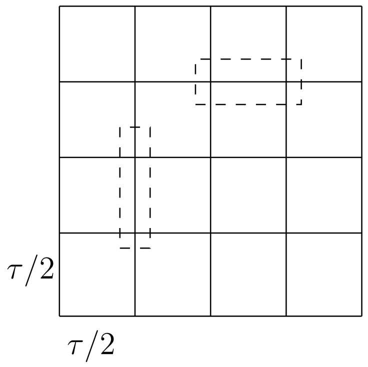

Figure 4 A rectangle R of perimeter 2τ can intersect at most six grid cells.

Computing the depth within a cell: Rectangles fully containing the cell. Consider the problem of computing for each such cell $C_{i,j}$, a counter $w_{i,j}$, equal to the number of rectangles of $\mathcal{R}$ that fully contain $C_{i,j}$. We compute the $w_{i,j}$'s by a sweep over the vertical slabs, maintaining a dynamic segment tree (see [3]) over the horizontal slabs, or rather cells, within the currently swept vertical slab. When moving from one vertical slab to the next, we find the $w_{i,j}$'s of all cells in the new slab, in the following three steps. First, we remove all the vertical segments that correspond to rectangles with a right edge in the current vertical slab (since these rectangles do not contribute to the $w_{i,j}$'s in this slab). Then, we traverse the segment tree bottom-up and compute, for each leaf, the number of segments (the y-spans of the active rectangles) containing the cell that corresponds to the leaf, thereby obtaining the desired counters $w_{i,j}$. We then add the segments that correspond to rectangles with a left edge in the current vertical slab (provided that their right edge is not in the slab). We have $O(n)$ insertions and deletions to/from this segment tree, each taking $O(\log n)$ amortized time. In addition, the number of nodes in the segment tree is $O(\sqrt{n})$ and we traverse it $\sqrt{n}$ times, to compute the counters $w_{i,j}$. It follows that the sweep takes $O(n \log n)$ total time.

Rectangles that straddle the cell. Let R be a rectangle that intersects $C_{i,j}$ (without fully containing it) and is not one of the $O(v_{i,j})$ rectangles that have a corner in $C_{i,j}$. Then R crosses $C_{i,j}$ in either a horizontal strip or a vertical strip; that is, $C_{i,j} \cap \partial R$ consists either of portions of one or two horizontal edges of R that cross $C_{i,j}$ from side to side, or of portions of one or two vertical edges that cross $C_{i,j}$ from side to side. Let $H_{i,j}$ (resp., $V_{i,j}$) denote the set of the horizontal (resp., vertical) edges of this kind, and let $\mathcal{R}^H_{i,j}$ (resp., $\mathcal{R}^V_{i,j}$) denote the set of the rectangles that contain the edges of $H_{i,j}$ (resp., of $V_{i,j}$). See Figure 2. The edges in $H_{i,j}$ partition $C_{i,j}$ into at most $2\sqrt{n}$ horizontal strips, and the depth within each strip, with respect to the rectangles in $\mathcal{R}^H_{i,j}$, is fixed, and changes by ± 1 as we move from one strip to the next. Analogous properties hold for the edges in $V_{i,j}$.

It follows that if we ignore the $O(v_{i,j})$ rectangles with vertices in $C_{i,j}$ (we refer to them as *short rectangles*) and the rectangles that fully contain $C_{i,j}$, the maximum depth in $C_{i,j}$ is the maximum depth $\delta^H_{i,j}$ of the rectangles of $\mathcal{R}^H_{i,j}$, plus the maximum depth $\delta^V_{i,j}$ of the rectangles of $\mathcal{R}^V_{i,j}$. Each of $\delta^H_{i,j}$, $\delta^V_{i,j}$ is the maximum of a sequence of depths, of length at most $2\sqrt{n}$, where consecutive elements differ by ± 1.

We maintain the vertical projections (i.e., y-spans) of the intersections of the rectangles of $\mathcal{R}^H_{i,j}$ with $C_{i,j}$ in a segment tree $T^H_{i,j}$ (which we will make dynamic in the next section).

Each leaf v of $T^H_{i,j}$ is associated with a strip $[h', h)$, where h' and h are consecutive edges of $H_{i,j}$. The depth of this strip (with respect to $\mathcal{R}^H_{i,j}$ is the sum of the numbers of segments stored at the ancestors of v).

We use this data structure to find, in $O(\log n)$ time, the strip of maximum depth in any subsequence of consecutive strips, by storing at each internal node v of $T^H_{i,j}$ the maximum depth of the leaves in its subtree (we omit the straightforward and fairly routine details of this mechanism).

We maintain $\mathcal{R}^V_{i,j}$ in an analogously defined dynamic segment tree $T^V_{i,j}$. Clearly, $T^H_{i,j}$ and $T^V_{i,j}$ allow us to answer maximum depth queries, with respect to $\mathcal{R}^H_{i,j} \cup \mathcal{R}^V_{i,j}$, where each query specifies a range of consecutive horizontal strips and a range of consecutive vertical strips, and asks for the maximum depth (with respect to $\mathcal{R}^H_{i,j} \cup \mathcal{R}^V_{i,j}$) within the rectangular Cartesian product of these ranges. Each such query takes $O(\log n)$ time.

Short rectangles and computing the maximum depth. We use the structures $T^H_{i,j}$ and $T^V_{i,j}$ to compute the real maximum depth within $C_{i,j}$, which also takes into account the $w_{i,j}$ rectangles that fully contain $C_{i,j}$, and the $O(v_{i,j})$ short rectangles. To do so, we partition $C_{i,j}$ into $O(v^2_{i,j})$ axis-parallel rectangular subcells by the horizontal and vertical lines that pass through the vertices inside $C_{i,j}$,[1] and query $T^H_{i,j}$ and $T^V_{i,j}$ for the maximum depth (in $\mathcal{R}^H_{i,j} \cup \mathcal{R}^V_{i,j}$) within each of the resulting subcells, to which we refer as *query subcells*. See Figure 3. (Note that, in general, the left and right boundary edges of each query subcell cross the interiors of two vertical strips stored in $T^V_{i,j}$, and the bottom and top boundary edges of each query subcell cross the interiors of two vertical strips stored in $T^H_{i,j}$. We expand the horizontal and the vertical ranges of the query subcell to fully include the four relevant strips.) Each query subcell, though, has an additional weight, equal to the number of short rectangles that fully contain the subcell (by construction, there is no partial overlap of any short rectangle with a query subcell). We refer to this additional weight of a query subcell as the *short weight* of the subcell. These short weights are easy to compute in $O(v^2_{i,j})$ time, by constructing the coarse grid of these subcells (within $C_{i,j}$), followed by a suitable traversal of the subcells, updating the count by 0, +1, or −1, as we pass from one subcell to the next. For each query subcell, we add this short weight, and the global counter $w_{i,j}$, to the depths returned by the queries to $T^H_{i,j}$ and $T^V_{i,j}$. We then output the maximum of the resulting depths, over all the $O(v^2_{i,j})$ query subcells.

Analysis. The running time of this algorithm is bounded as follows. We first sort the vertices of the rectangles by their x- and y-coordinates, and compute the global counters $w_{i,j}$. This initialization takes $O(n \log n)$ time. Then, for each cell $C_{i,j}$, we construct the trees $T^H_{i,j}$ and $T^V_{i,j}$, in $O(\sqrt{n})$ time (the relevant edges are already sorted), for an overall $O(n^{3/2})$ time. We then spend $O(v^2_{i,j})$ time for constructing the coarse grid of query subcells within $C_{i,j}$, and $O(v^2_{i,j} \log n)$ time for querying $T^H_{i,j}$ and $T^V_{i,j}$ with these subcells. Summing over all cells, and using the fact that $v_{i,j} \le 4\sqrt{n}$ for each cell $C_{i,j}$, we get a total time of $O\left(n^{3/2} + \sqrt{n}\sum_{i,j} v_{i,j} \log n\right) = O\left(n^{3/2} \log n\right)$.

[1] When constructing this partition into query subcells, we also consider vertices on the bottom and left edges of $\partial C_{i,j}$, because, by our convention, these vertices are considered to be internal to the cell.

2.2 Updating the data structure

In this section we show how to dynamically maintain the counters $w_{i,j}$'s, the trees $T^H_{i,j}$ and $T^V_{i,j}$, the number of short rectangles in $C_{i,j}$ containing each query subcell, and the depth of each query subcell, as we increase z from 0 to τ. To retrieve the maximum depth after updating the structures at each critical z-value, we also maintain the depths of the query subcells, over all cells $C_{i,j}$, in a priority queue, and keep track of the maximum value in this queue. The overall maximum depth, i.e., the maximum number of points of P contained in an axis-parallel rectangle of perimeter 2τ, is the maximum attained by this priority queue throughout the z-sweep.

Note that at $z = 0$ each rectangle is essentially a vertical line segment (The lines supporting the left and right edges of the same rectangle are identical, and formally they are regarded as consecutive in G.). This simplifies that initialization of the data structure. For example initially $w_{i,j} = 0$ for all $1 \le i, j \le \sqrt{n}$.

As the value of z increases, the coordinates of the vertices of the rectangles in $\mathcal{R}(z)$ vary continuously, and so do the coordinates of the vertical and horizontal supporting lines that form the grid $G(z)$. However, discrete changes in the structure of $G(z)$ occur only when two horizontal or two vertical sides of two distinct rectangles partially overlap, or, in the looser sense that we follow, when the lines supporting two such edges coincide. The maximum depth in $\mathcal{A}(z)$ can change only at these discrete events.

Consider an event where the right side of one rectangle R_1 and the left side of another rectangle R_2 swap their vertical order; that is, the two vertical lines supporting these edges in $G(z)$ coincide and then swap their order. The event where two horizontal sides partially overlap is handled in a fully symmetric fashion This swap takes place either within a single vertical slab B_j, or across the boundary ('divider') between two adjacent slabs. In total, this affects up to $4\sqrt{n}$ cells of G (within these slabs). We describe here the case in which the swap occurs within a single vertical slab B_j; the other case is handled in a similar manner, with a few minor modifications, and will appear in the full version of this paper.

Consider first the cells of B_j that do not contain the vertices of R_1 and of R_2. Within each such cell $C_{i,j}$, the effect of the swap is that the right endpoint of the horizontal segment corresponding to R_1 and the left endpoint of the horizontal segment corresponding to R_2 swap their order in $T^V_{i,j}$, assuming they both cross $C_{i,j}$. As a result, the depth of a single vertical strip with respect to $\mathcal{R}^V_{i,j}$ increases by 2. This is because the x-spans of the rectangles in $\mathcal{R}(z)$ increase as z increases; in the symmetric case of horizontal strips, the opposite holds – the depth of a single strip decreases by 2. (As already mentioned, this holds only for cells within the common y-range of R_1 and R_2; no change occurs in the other cells.) We update the corresponding segments in the tree $T^V_{i,j}$. This takes $O(\log n)$ amortized time per cell, and $O(\sqrt{n}\log n)$ time for all cells that are affected by the swap.

We now need to reapply the rectangular depth queries for the query subcells within each affected cell $C_{i,j}$, but we note that only $O(v_{i,j})$ of the queries can change their output – these are the queries whose subcells are crossed by the vertical strip that has changed its depth. We perform these queries, as in the static case, and update the depths of these query subcells in the global priority queue accordingly ($w_{i,j}$, and the short weights of the affected subcells, do not change).

Consider next the at most four cells $C_{i,j}$ that contain vertices of R_1 or of R_2 (that is, endpoints of the swapped vertical edges). Here the swap does not affect $T^V_{i,j}$, because only one (or none, when endpoints of both edges lie in $C_{i,j}$) of the swapped vertical edges belongs to this set. We split the rest of the description of the required updates according to whether $C_{i,j}$ contains vertices only of R_1 or only of R_2, or vertices of both R_1 and R_2.

Consider first the case where $C_{i,j}$ contains only vertices of R_1, either the top-right vertex, or the bottom-right vertex, or both; $C_{i,j}$ may also contain left vertices of R_1, but they have no effect on the update procedure. The case where $C_{i,j}$ contains only the top-left vertex or the bottom-left vertex of R_2, or both (and maybe also right vertices), is handled in a fully symmetric manner. Denote by λ_1^R (resp., λ_2^L) the vertical line supporting the right side of R_1 (resp., the left side of R_2); these are the lines that swap their left-to-right order. In this case the partition of $C_{i,j}$ into query subcells essentially does not change, except that one line in $V_{i,j}$, namely λ_2^L (it is in $V_{i,j}$ because the left vertices of R_2, which it supports, are not in $C_{i,j}$) moves from one column of query subcells (just to the right of λ_1^R) to another column (just on the left of that line). We query $T_{i,j}^H$ and $T_{i,j}^V$ to get the new depth of each of the query subcells to the left and to the right of λ_1^R, with respect to the rectangles in $\mathcal{R}_{i,j}^H \cup \mathcal{R}_{i,j}^V$, add to it $w_{i,j}$ and the short weight of the subcell, and update the depths of all these subcells in the priority queue.

If $C_{i,j}$ contains at least one vertex of R_1 and at least one vertex of R_2 then, in the grid defining the partition of $C_{i,j}$ into query subcells, the corresponding vertical lines λ_1^R and λ_2^L swap (with the former moving to the right of the latter). Consequently, the short weights of some of the subcells in the column bounded these two lines (which 'closes' at the swap and 're-opens' afterwards) increases by 2; the affected cells are those that lie in the overlap between the y-spans of R_1 and R_2 (as before, the corresponding short weights decrease by 2 in the symmetric case of a horizontal swap). We locally update these short weights and the depths of these subcells accordingly, and similarly update the priority queue.

In both cases, the number of affected query subcells of $C_{i,j}$ is only $O(v_{i,j})$, so the total amortized update time is $O(v_{i,j}\log n)$.

The overall amortized time spent on maximum depth queries in $T_{i,j}^H$ and in $T_{i,j}^V$, and on updates of short weights, is $O\left(\sum_i v_{i,j}\log n\right)$, where we sum over all cells $C_{i,j}$ in the single vertical column B_j. Fortunately, $\sum_i v_{i,j} \le 4\sqrt{n}$, so the overall (amortized) cost of an update is $O\left(\sqrt{n}\log n\right)$.

As we mentioned the case where the swap occurs across a horizontal or a vertical divider between adjacent is similar and takes the same amortized time. This completes the description and analysis of the data structure, including both correctness and performance bounds, and justifies the bounds given in Theorem 1.

2.3 An output-sensitive algorithm

Let k be the maximum number of points of P in an axis-parallel rectangle of perimeter 2τ. In this section we show how to modify our algorithm so that it runs in $O(nk^{3/2}\log k)$ time. The same modification holds for the case of fixed diagonal, but not for fixed area.

We cover the plane by a grid whose cells are of size $\tau/2 \times \tau/2$, and count the number of points of P in each nonempty cell. Using the floor function and a universal hashing scheme (see, e.g., [4]), this takes $O(n)$ expected time. Let k_0 denote the maximum number of points in any grid cell. It follows that $k_0 \le k \le 6k_0$, where the left inequality follows since each grid cell has perimeter 2τ, and the right inequality follows since any rectangle R of perimeter 2τ (and in particular the optimal one) can intersect at most six grid cells as is easily checked; see Figure 4.

We collect, in $O(n)$ time, all the 2×3 and 3×2 clusters C of grid cells such that C contains at least k_0 points of P, and observe that the number of such clusters is $O(n/k_0) = O(n/k)$. We apply our algorithm to each cluster separately and return the rectangle containing the largest number of points in any of the clusters. We thus obtain the following theorem.

▶ **Theorem 2.** *Given a set P of n points in the plane in general position, and a parameter $\tau > 0$, let k denote the maximum number of points of P in an axis-parallel rectangle of perimeter 2τ. One can find, in $O((n/k)k^{5/2}\log k) = O(nk^{3/2}\log k)$ time, an axis-parallel rectangle of perimeter 2τ that contains the maximum number k of points of P. The algorithm requires $O(n + k^{3/2}\log k)$ storage.*

▶ Remark.

(1) The technique in this subsection also works for finding an axis-parallel rectangle with diagonal of length d containing the maximum number of points of P. We use a $(d/\sqrt{2}) \times (d/\sqrt{2})$ grid (each of whose cells has diagonal $= d$), argue that any axis-parallel rectangle of diagonal d is contained in some small local cluster of grid cells, and obtain, as above, an algorithm that runs in $O(nk^{3/2}\log k)$ time, where k is the maximum number of points in a rectangle of diagonal d.

(2) The technique in this subsection does not extend to the case of rectangles with a fixed area, since no single grid can localize every rectangle of area A within a small cluster of its cells. (In contrast, as already noted, the main algorithm, which runs in $O(n^{5/2}\log n)$ time, does extend to the case of fixed area.)

Nevertheless, for the case of a fixed area, say A, we can use a similar idea to get an algorithm whose running time depends (albeit rather weakly) on A, as follows. Assume that $P \subset [0,1]^2$, and that the given area is $A < 1$ (the case $A \geq 1$ is clearly trivial). Without loss of generality, it suffices to consider only rectangles of width between A and 1, with the corresponding height between 1 and A. We can partition the problem into $O(\log(1/A))$ subproblems, so that in each subproblem we only consider rectangles whose widths are between z_0 and $2z_0$, and heights between A/z_0 and $A/(2z_0)$, for some fixed z_0. To each subproblem we can apply a suitable variant of the preceding grid construction, and solve the subproblem in $O(nk^{3/2}\log k)$ time, for a total cost of $O(nk^{3/2}\log k\log(1/A))$ time. We leave it as an open problem to obtain an algorithm whose running time bound is k-sensitive and independent of A, in the style of Theorem 2.

3 An approximate solution

In this section we present a randomized algorithm that computes, with high probability, a rectangle of perimeter 2τ that contains at least $(1-\varepsilon)k$ points of P, for a prescribed $0 < \varepsilon < 1$, where k is the maximum possible value, and runs in time $O\left(n + \frac{n}{k\varepsilon^5}\log^{5/2}\frac{n}{k}\log\left(\frac{1}{\varepsilon}\log\frac{n}{k}\right)\right)$.

We use the grid partitioning of Section 2.3, and obtain (i) an approximation k_0 of k, up to a factor 6, and (ii) $O(n/k_0) = O(n/k)$ clusters of points, each of size $\Theta(k)$. We apply the following procedure to each cluster separately, and return the rectangle containing the largest number of points of P, among those output for each of the clusters.

So let C be a fixed "heavy" cluster of grid cells, and let $P_C = P \cap C$ denote the set of points of P in C (of size $\Theta(k)$). We take a random sample S of size $s = \Theta\left(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\right)$, for some $0 < \delta < 1$. For a suitable sufficiently large constant of proportionality, S is an $(\varepsilon/12)$-*approximation* of P for axis-parallel rectangular ranges, with probability at least $1-\delta$. That is, with probability $\geq 1-\delta$, we have, for each axis-parallel rectangle R,

$\left|\frac{|R\cap S|}{|S|}-\frac{|R\cap P_C|}{|P_C|}\right|\le\frac{\varepsilon}{12}$ (see, e.g., [11]).[2]

We now run the exact algorithm of Section 2.3 on S, and obtain an axis-parallel rectangle R_S (of perimeter 2τ) that contains the maximum number of points of S.

Correctness. Let R be an optimum rectangle (of perimeter 2τ) that contains k points of P, and let C be a cluster that fully contains R; by the arguments in Section 2.3, such a cluster always exists. Let S be the corresponding random sample of s points of P_C, and let R_S denote, as above, the axis-parallel rectangle of perimeter 2τ that contains the maximum number of points, denoted s^*, of S. Then, with probability $\ge 1-\delta$, we have $\left|\frac{|R\cap S|}{s}-\frac{k}{t}\right|\le\frac{\varepsilon}{12}$, where $t=|P_C|=\Theta(k)$. On the other hand,

$$\left|\frac{|R_S\cap S|}{s}-\frac{|R_S\cap P_C|}{t}\right|=\left|\frac{s^*}{s}-\frac{|R_S\cap P_C|}{t}\right|\le\frac{\varepsilon}{12},$$

$$\text{that is,}\quad |R_S\cap P_C|\ge\frac{s^*t}{s}-\frac{1}{12}\varepsilon t\ge\frac{|R\cap S|t}{s}-\frac{1}{12}\varepsilon t\ge\left(k-\frac{1}{12}\varepsilon t\right)-\frac{1}{12}\varepsilon t=k-\frac{1}{6}\varepsilon t.$$

Since $t\le 6k$, this is $\ge(1-\varepsilon)k$. That is, the procedure will find, with probability at least $1-\delta$, a rectangle that contains at least $(1-\varepsilon)k$ points of P.

Running time. Finding the heavy clusters C takes a total of $O(n)$ time, and the overall cost of drawing the random samples S also takes $O(n)$ time. We take $\delta=(k/n)^c$, for some suitably large exponent c, so as to guarantee correctness with high probability in all clusters. (There are only $O(n/k)$ clusters, so the probability to fail in at least one of them is at most $O((n/k)\cdot(k/n)^c)=O((k/n)^{c-1})$.) The cost of a single application of the exact algorithm is $O\left(\frac{1}{\varepsilon^5}\log^{5/2}\frac{n}{k}\log\left(\frac{1}{\varepsilon}\log\frac{n}{k}\right)\right)$. Since the number of heavy clusters is $O(n/k)$, we obtain a total running time of $O\left(n+\frac{n}{k\varepsilon^5}\log^{5/2}\frac{n}{k}\log\left(\frac{1}{\varepsilon}\log\frac{n}{k}\right)\right)$. Assuming that k is not too small and neglecting the logarithmic terms, we can even take $\varepsilon=1/k^{1/5}$, and obtain an algorithm whose running time is $\approx O(n)$, which returns, with high probability, an axis-parallel rectangle of perimeter 2τ that contains at least $k-O(k^{4/5})$ points of P.

The same technique applies with minor modifications also to the case of a fixed diagonal. We summarize the results of this section in the following theorem.

▶ **Theorem 3.** *Let P be a set of n points in the plane, and let $\tau>0$ and $\varepsilon\in(0,1)$ be given parameters. We can compute, in time $O\left(n+\frac{n}{k\varepsilon^5}\log^{5/2}\frac{n}{k}\log\left(\frac{1}{\varepsilon}\log\frac{n}{k}\right)\right)$, an axis-parallel rectangle of perimeter 2τ that contains at least $(1-\varepsilon)k$ points of P, where k is the maximum number of points contained in such a rectangle. Within the same time bound we can compute an axis-parallel rectangle of diagonal d that contains at least $(1-\varepsilon)k$ points of P, where k is the maximum number of points contained in such a rectangle.*

We note that for the case of a fixed area, an approximation algorithm, based on a totally different approach, was recently obtained by De Berg et al. [6]; it runs in $O((n/\varepsilon^4)\log^2 n\log(1/\varepsilon))$ time.

[2] Using discrepancy based methods one can find a smaller ε-approximation of size $O(\frac{1}{\varepsilon}\log^2(\frac{1}{\varepsilon}))$ [2]. However, such an ε-approximation is less efficient to compute (although polynomial).

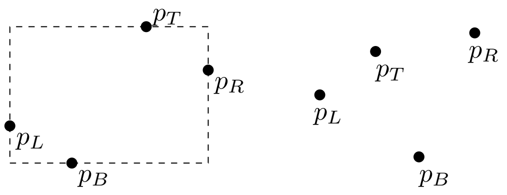

Figure 5 A quadruple (p_L, p_R, p_B, p_T) that defines a valid rectangle (left), and a quadruple that does not (right).

4 An efficient exact algorithm for min-perim(k)

In this section we present an efficient algorithm for the dual version of the problem, in which we specify k, and seek an axis-parallel rectangle of smallest perimeter that contains k points of P. The same technique also applies to the cases where the objective is to minimize the area, or the diagonal, of the enclosing rectangle. For the case of maximum diagonal, we obtain the same bound, and for the case of maximum area, we obtain a bound that is not output sensitive. In what follows we focus on the case of minimum perimeter, and only later discuss the extensions to the cases of minimum area or diagonal.

Let Q be an optimum rectangle, namely, an axis-parallel rectangle of smallest perimeter that contains k points of P. Clearly, each side of Q must contain a point of P, where these four points are not necessarily distinct (the number of distinct points is always between two and four). Denote by p_L, p_R, p_B, and p_T the points that lie on the left, right, bottom, and top sides of Q, respectively. Naively, there are $\binom{n}{2}$ candidate pairs (p_L, p_R), and $\binom{n}{2}$ candidate pairs (p_B, p_T). However, using an observation of [7], the number of candidate pairs is only $O(nk)$, because p_R is one of the $O(k)$ rectilinear nearest neighbors of p_L, and similarly for p_T and p_B. We find the $O(nk)$ left-right candidate pairs, and the $O(nk)$ bottom-top candidate pairs, in $O(n \log n + nk)$ time, as in [7]. We sort the ordered pairs (p_L, p_R) of distinct points of P, with p_L lying to the left of p_R, in increasing order of the differences between their x-coordinates, into a list X, and apply a symmetric construction with respect to the bottom-top pairs (p_B, p_T) and their y-coordinates, to obtain another sorted list Y.

Consider the matrix M whose rows are the elements of X (in sorted order) and whose columns are the elements of Y (in sorted order). For each pair of pairs $\pi_1 = (p_L, p_R)$ of X and $\pi_2 = (p_B, p_T)$ of Y, put $M(\pi_1, \pi_2) = (x(p_R) - x(p_L)) + (y(p_T) - y(p_B))$, and note that $M(\pi_1, \pi_2)$ is half the perimeter of the axis-parallel rectangle defined by the quadruple (p_L, p_R, p_B, p_T). To be precise, not every such quadruple defines a valid rectangle, but each valid candidate rectangle is defined by such a quadruple; See Figure 5.

The matrix M is a monotone matrix, that is, each of its rows and each of its columns is sorted in increasing order. Using the algorithm of Frederickson and Johnson [10] (see also [8, 9, 13]), We can find the ρ-th largest element in M in time $O(nk)$, for any rank ρ.

We thus run a binary search through the $O((nk)^2)$ critical perimeters (that is, entries of M), by making $O(\log n)$ calls to our algorithm MAX-PTS(τ), where the outcome of each call guides the continuation of the binary search. Each call incurs an overhead of $O(nk)$ time to find in M the relevant perimeter τ, and the algorithm itself takes time $O(nk_\tau^{3/2} \log k_\tau)$ where k_τ is the maximum number of points in a rectangle of perimeter 2τ (see Theorem 2). To make the overall running time bound k-sensitive, we pause the execution of the algorithm for MAX-PTS(τ) after the step where it obtains an approximation k_0 to k_τ, satisfying $k_0 \le k_\tau \le 6k_0$. If $k_0 > k$ we know that $k_\tau > k$, and we continue the binary search with a smaller τ. If $k > 6k_0$ we know that $k > k_\tau$, and we continue the binary search with a larger

τ. If $k_0 \le k \le 6k_0$, we let the algorithm run to completion, and bifurcate depending on the relation between the output k_τ and k. Altogether, we obtain the following result.

▶ **Theorem 4.** *Given a set P of n points in the plane, and a parameter $k \le n$, we can find an axis-parallel rectangle of minimum perimeter that contains k points of P in time*

$$O\left(nk^{3/2}\log k \log n\right).$$

▶ Remark. The same procedure applies to the cases of minimum area or minimum diagonal. For the case of diagonal, we obtain the same performance bound. For the case of area, we have to put all $\binom{n}{2}$ pairs of points in X and Y, so selection in M takes $O(n^2)$ time, and the "decision procedure" MAX-PTS(A), where A is the given area, now takes $O(n^{5/2}\log n)$ time, resulting in a k-insensitive algorithm that runs in time $O(n^{5/2}\log^2 n)$. This is still a significant improvement over the recent algorithm of De Berg et al. [6] when k is not too small. It is an open problem to get an output sensitive bound, similar to the one in Theorem 4, for the case of area.

References

1 A. Aggarwal, H. Imai, N. Katoh, and S. Suri. Finding k points with minimum diameter and related problems. *J. Algorithms*, 12(1):38–56, 1991.

2 N. Bansal and S. Garg. Algorithmic discrepancy beyond partial coloring. In *Proc. of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 914–926, 2017.

3 Y-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9):1412–1434, 1992.

4 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

5 A. Datta, H.P. Lenhof, C. Schwarz, and M. Smid. Static and dynamic algorithms for k-point clustering problems. *J. Algorithms*, 19(3):474–503, 1995.

6 M. de Berg, S. Cabello, O. Cheong, D. Eppstein, and C. Knauer. Covering many points with a small-area box. *CoRR*, abs/1612.02149, 2016.

7 D. Eppstein and J. Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discrete Comput. Geom.*, 11(3):321–350, 1994.

8 G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in X + Y and matrices with sorted columns. *J. Comput. Syst. Sci.*, 24(2):197–208, 1982.

9 G. N. Frederickson and D. B. Johnson. Finding kth paths and p-centers by generating and searching good data structures. *J. Algorithms*, 4(1):61–80, 1983.

10 G. N. Frederickson and D. B. Johnson. Generalized selection and ranking: Sorted matrices. *SIAM J. Comput.*, 13(1):14–30, 1984.

11 S. Har-peled. *Geometric Approximation Algorithms*. American Mathematical Society, Boston, MA, USA, 2011.

12 H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *J. Algorithms*, 4(4):310–323, 1983.

13 A. Mirzaian and E. Arjomandi. Selection in X + Y and matrices with sorted rows and columns. *Information Processing Letters*, 20(1):13–17, 1985.

14 S.C. Nandy and B.B. Bhattacharya. A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids. *Computers Math. Applic.*, 29(8):45–61, 1995.

15 M. Segal and K. Kedem. Enclosing k points in the smallest axis parallel rectangle. *Inform. Process. Letts.*, 65(2):95–99, 1998.