Space-efficient Algorithms for Empty Space Recognition among a Point Set in 2D and 3D

Minati De*† Subhas C. Nandy*

Abstract

In this paper, we consider the problem of designing in-place algorithms for computing the maximum area empty rectangle of arbitrary orientation among a set of points in 2D, and the maximum volume empty axis-parallel cuboid among a set of points in 3D. If n points are given in an array of size n, the worst case time complexity of our proposed algorithms for both the problems is $O(n^3)$; both the algorithms use $O(1)$ extra space in addition to the array containing the input points.

1 Introduction

Designing low memory algorithms is considered to be an important paradigm for the data-streaming and data-mining applications. Here the amount of data available is huge, and it is wise to consider as much data as possible to get precise result. In other areas also, the low memory algorithms are important in spite of the fact that computer hardware has become extremely cheap now-a-days. For an example, consider the VLSI physical design applications, where the number of circuit modules in a VLSI chip are rapidly growing day by day, and running the standard routing, placement, verification algorithms, are becoming impossible even in the modern computers due to the size of data. In sensor network applications, it is often found that in order to get precise information, a huge number of sensors are deployed. Moreover in tiny devices, for example, sensors, GPS systems, mobile hand-sets, small robots, etc, in order to maintain its size, one needs to lower down the memory size. For all these reasons, designing low-memory algorithms for practical problems have become a challenging task to the algorithm researchers.

In computational geometry, in-place algorithms are studied for a very few problems. For the convex hull problem in 2D, the best known result is an $O(n \log h)$ algorithm with $O(1)$ extra space [5]. Bronnimann et al. [4] showed that the upper hull of a set of n points in 3D can be computed in $O(n \log^3 n)$ time using $O(1)$ extra space. The best known algorithm for this problem runs in $O(n \log n)$ expected time [8]. Bose et al. [3] used an in-place divide and conquer technique to solve the following problems in 2D using $O(1)$ extra space: (i) a deterministic $O(n \log n)$ time algorithm for the closest pair problem, (ii) a randomized expected $O(n \log n)$ time algorithm for the bichromatic closest pair problem, and (iii) a deterministic $O(n \log n+k)$ time algorithm for computing the intersections among orthogonal line segments. For computing the intersections among arbitrary line segments, two algorithms are available in [7]. If the input array can be used for storing intermediate results, then the problem can be solved in $O((n+k) \log n)$ time and $O(1)$ space. but, if the input array is not allowed to be destroyed, then the time complexity increases by a factor of $\log n$; it also requires $O(\log^2 n)$ extra space. Vahrenhold [15] proposed an $O(n^{\frac{3}{2}} \log n)$ time and $O(1)$ extra space algorithm for the Klee's measure problem, where the objective is to compute the union of n axis-parallel rectangles of arbitrary sizes. Asano and Rote [2] showed that all the Delaunay triangles among a set of n points can be computed in $O(n^2)$ time using $O(1)$ space. This, in turn, recognizes the largest empty circle among a point set with the same time complexity.

We will now consider the algorithms for recognizing the *maximum area empty rectangle* among a set of n points in a region $\mathcal{R}$ in 2D. The axis-parallel version of the problem was first introduced by Namaad et al. [13]. They introduced the concept of *maximal empty rectangle* (MER). It is an empty rectangle, not properly contained in any other empty rectangle. They showed that the number of MERs (m) among a set of n points may be $\Omega(n^2)$ in the worst case; but if the points are randomly placed, then the expected value of m is $O(n \log n)$. In the same paper, an $O(min(n^2, m \log n))$ time algorithm for identifying the largest MER was also proposed. Orlowski [14] proposed an $O(m + n \log n)$ time algorithm for finding the largest MER that inspects all the MERs present in $\mathcal{R}$, and identifies the largest one. The best known algorithm for this problem runs in $O(n \log^2 n)$ time in the worst case [1]. All these algorithms use $O(n)$ extra space. The worst case time and space complexities for computing the largest empty rectangle of arbitrary orientation among a set of n points are $O(n^3)$ and $O(n^2)$ respectively [6]. Recently, an in-place algorithm for recognizing the largest empty axis-parallel rectangle is proposed that runs in $O((m+n) \log n)$ time and uses $O(1)$ extra space in ad-

*Indian Statistical Institute, Kolkata, India.
†Presently visiting Carleton University, Canada. minati.isi@gmail.com

dition to the array containing the input points [9]. It uses a novel way of maintaining priority search tree in an in-place manner. In 3D, the largest empty axis-parallel cuboid among a set of n points in an axis-parallel cuboid region $\mathcal{R}$ can be computed in $O(C + n^2 \log n)$ time with $O(n)$ extra space, where C is the number of maximal empty axis-parallel cuboids in $\mathcal{R}$, which may be $O(n^3)$ in worst case [12].

We first describe an in-place algorithm for computing the maximum area empty rectangle of any arbitrary orientation among a set of n points in a 2D rectangular region. We will also consider a simplified 3D version of the problem, where the objective is to identify the maximum volume empty axis-parallel cuboid among a set of n points in a 3D axis-parallel region. The time complexity of both the algorithms is $O(n^3)$, and they need $O(1)$ space in addition to the input array.

2 Computing largest MER of arbitrary orientation

We now propose an in-place algorithm for finding maximum area empty rectangle of arbitrary orientation among a set of points P inside a rectangular region $\mathcal{R}$. The problem was addressed by Chaudhuri et al. [6]. They introduced the concept of PMER. A PMER, defined by four points $p_i, p_j, p_k, p_\ell \in P$, is the maximum area rectangle of any arbitrary orientation whose four sides pass through p_i, p_j, p_k and p_ℓ, and the interior of the rectangle does not contain any member of P. It is shown that the number of PMERs is bounded above by $O(n^3)$. It follows from the following observation:

Observation 1 *[6] At least one side of a PMER must contain two points from P, and other three sides either contain at least one point of P or the boundary of $\mathcal{R}$.*

2.1 Algorithm

Observation 1 plays the central role in our algorithm. We consider each pair of points $p, q \in P$, and compute all the PMERs with one side passing through p, q. We use geometric duality for solving this problem. The duality transform in 2D maps a point $p = (\alpha, \beta)$ in the primal plane into a line $p' = \alpha x - \beta$ in the dual plane and maps a non-vertical line $\ell : y = mx - c$ in the primal plane into the point $\ell' = (m, c)$ in the dual plane. For the standard properties of duality transform, see [10].

Observation 2 *Let v be a point on a vertical line $\mathcal{L}$ in the dual plane, and $q'_1, q'_2, \ldots, q'_m$ be m lines in the dual plane that intersect $\mathcal{L}$ in one side (above or below) of v, and are arranged in increasing order of their distances from v along $\mathcal{L}$. Now, all the points $q_1, q_2, \ldots q_m$ are in one side (below or above) of the line v' in the primal plane and the perpendicular distances of $q_1, q_2, \ldots, q_m$ from the line v' are also in increasing order.*

We will consider the arrangement $\mathcal{A}(P)$ of the set of dual lines corresponding to all the points in the array P. Its each vertex v_{ij} obtained by the intersection of the dual lines p'_i and p'_j, corresponds to the line ℓ_{ij} passing through $p_i, p_j \in P$ in the primal plane. Thus, in order to get the lines passing through each pair of points in P, we need to visit all the vertices in $\mathcal{A}(P)$.

Note that, each element of P corresponding to an input point also represents the corresponding dual line. We first identify the left-most vertex in $\mathcal{A}(P)$ by computing the intersections of all the $O(n^2)$ pairs of dual lines. Now, a vertical line $\mathcal{L}$ starts sweeping from that position. We execute a sorting step to arrange the members in P such that the y-coordinates of the points of intersection of those dual lines and the sweep line $\mathcal{L}$ are in increasing order. Thus, P also serves the role of the sweep line status array. During the sweep, this property of P is always maintained. Here the two lines, say p' and q', incident to the next vertex $v \in \mathcal{A}(P)$ will remain consecutive, say at $P[i]$ and $P[i+1]$. All the lines below (resp. above) v are to the right of $P[i+1]$ (resp. left of $P[i]$) in the array P, and are in increasing order of their distances from the point v along the line $\mathcal{L}$. We process v to compute all the MERs whose one side passes through (p, q) using the procedure **process**(p, q). The procedure **get_next_vertex** computes the next vertex of $\mathcal{A}(P)$ that $\mathcal{L}$ faces to the right of v during the sweep.

2.1.1 get_next_event

After processing a vertex v (intersection of a pair of dual lines p' and q' stored at $P[i]$ and $P[i+1]$ respectively), when $\mathcal{L}$ moves to the right of v, p' and q' are swapped in P for maintaining their order along $\mathcal{L}$. We do not maintain the event queue. At each step, we compute the next vertex in $\mathcal{A}(P)$ to be processed.

Observation 3 *[11] At any instant of time during the sweep, the vertex closest to $\mathcal{L}$ to its right side is the point of intersection of a pair of dual lines that are consecutive in the ordered list of dual lines.*

We compute the intersection of each pair of consecutive dual lines in the array P. If it is to the right of $\mathcal{L}$, then it is a *feasible intersection point* (FIP). By Observation 3, The next vertex of $\mathcal{A}(P)$ to the right of $\mathcal{L}$ corresponds the left-most FIP. If no such FIP is obtained, the sweep stops. Thus, the time complexity for getting the next vertex of $\mathcal{A}(P)$ for processing is $O(n)$.

2.1.2 Process(p, q)

Let v be the vertex in $\mathcal{A}(P)$ under process. It corresponds to the pair of points $p, q \in P$ stored at $P[i]$ and $P[i+1]$ respectively. Let λ be the straight line passing through p, q. By Observation 2 the points below λ are $\Pi_1 = \{P[i+2], P[i+3], \ldots, P[n]\}$ in increasing order of their distances from λ. We now describe the method of computing all the PMERs with (p, q) at its top boundary. The method of computing all the PMERs with (p, q) at their bottom boundary with the points $\Pi_2 = \{P[i-1], P[i-2], \ldots p[1]\}$ is the same.

Our algorithm considers a curtain whose two sides are bounded by the boundary of $\mathcal{R}$, and top boundary is attached to both p, q. The curtain falls in a manner parallel to the line λ. As soon as it hits a point $a \in \Pi_1$ it reports a PMER. This point is easily obtained from the sorted list Π_1. If the projection a^* of the point a on λ lies inside the interval $[p, q]$, the processing of λ stops. Otherwise, the curtain is truncated at a^*, and the process continues to process the next point in Π_1.

2.2 Complexity analysis

We have considered all the $O(n^2)$ vertices of $\mathcal{A}(P)$. Generation of each vertex v needs $O(n)$ time with $O(1)$ additional space. The time required for processing the vertex v for computing all the PMERs with one side passing through the pair of points (p, q) corresponding to the vertex v is also $O(n)$, and it needs $O(1)$ extra work-space. The algorithm needs to maintain a global counter to store the maximum area/perimeter PMER.

Theorem 1 *Given an array with n points, the maximum area/perimeter rectangle of arbitrary orientation can be computed in $O(n^3)$ time with $O(1)$ extra space.*

Corollary 1.1 *The method proposed in* **process**(p, q) *can also be used to compute the largest empty axis-parallel rectangle in $O(n^2)$ time.*

Proof. For computing the largest empty axis-parallel rectangle, we need not have to consider the duals of the points in P. Here for each point $p_i \in P$, we need to execute four line sweep passes as follows:

• Sweep a horizontal line upwards (resp. downwards) to get the largest axis-parallel MER with bottom (resp. top) boundary passing through p_i, and

• Sweep a vertical line towards left (resp. right) to get the largest axis-parallel MER with right (resp. left) boundary passing through p_i.

To execute the horizontal (resp. vertical) line sweep for all the points, we need to sort the points in P with respect to their y-coordinates (resp. x-coordinates) once only. Then the time complexity of the line sweep for each point $p_i \in P$ is $O(n)$. □

3 Computing largest axis-parallel MEC

We now describe the method of computing the largest empty cuboid among a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in a 3D axis-parallel parallelopiped (cuboid) $\mathcal{R}$ bounded by six axis-parallel planes. The coordinate of the point p_i is denoted by (x_i, y_i, z_i). A *maximal empty cuboid* (MEC) is a cuboid whose each face either coincides with a face of $\mathcal{R}$ or passes through a point in P, and its interior does not contain any point in P. The objective is to identify an MEC of maximum volume. There are three types of MECs' inside $\mathcal{R}$.

type-1: the MEC with both top and bottom faces aligned with the top and bottom faces of $\mathcal{R}$,

type-2: the MEC whose top face is aligned with the top face of $\mathcal{R}$, but bottom face passes through a point in P, and

type-3: the MEC whose top face passes through some point in P. Bottom face may pass through a point in P or may coincide with the bottom face of $\mathcal{R}$.

Theorem 2 *[12] The number of* type-1, type-2 *and* type-3 *MECs' inside $\mathcal{R}$ are $O(n^2)$, $O(n^2)$ and $O(n^3)$ respectively in the worst case.*

From now onwards, we use P to denote the array of size n containing the input points. We show that the methods proposed in [12] for identifying the largest *type-1*, *type-2* and *type-3* MECs can be made in-place with $O(1)$ extra work-space in addition to the input array.

3.1 Computation of largest *type-1* MEC

Consider the projections of the points in P on the top face H of $\mathcal{R}$. Note that, each maximal empty axis-parallel rectangle (MER) on H corresponds to a *type-1* MEC. Since the height of all these MECs' are the same, the problem reduces to computing the maximum area MER in H. Corollary 1.1 suggests the following result:

Lemma 3 *The largest empty* type-1 *MEC can be computed in $O(n^2)$ time using $O(1)$ extra work-space.*

3.2 Computation of largest *type-2* MEC

We assume that the points in P are sorted in decreasing order of their z-coordinates. We consider each point $p_i \in P$ in order, and compute $MEC(p_i)$, the largest *type-2* MEC whose bottom face passes through p_i. Let $H(p_i)$ be the horizontal plane passing through p_i, and

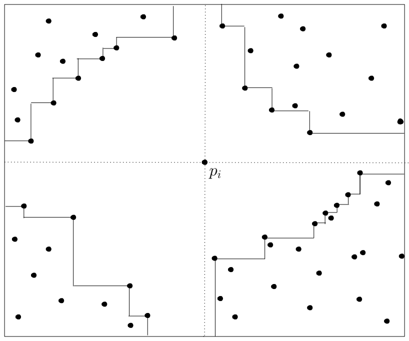

Figure 1: Empty orthoconvex polygon around p_i

$P_i = \{p_1, p_2, \ldots, p_k\}$ be the set of points strictly above $H(p_i)$. Note that, $MEC(p_i)$ corresponds to the largest MER on $H(p_i)$ containing the point p_i among the projections of the points in P_i on $H(p_i)$ as obstacles.

Let us partition the plane $H(p_i)$ into four quadrants by drawing two mutually perpendicular axis-parallel lines passing through p_i. In $O(n)$ time, we will be able to partition the portion of the array $P[1, 2, \ldots, k]$ into four parts, namely P_i^θ, $\theta = 1, 2, 3, 4$, where P_i^θ denote the points in the θ-th quadrant. The members in P_i^θ are in consecutive positions in the array P.

In each quadrant θ, we define the maximal closest stair $STAIR_\theta$ around p_i with a subset of points of P_i^θ as in [12]. $STAIR_\theta$ is unique in the θ-th quadrant. The concatenation of these four stairs describe an empty axis-parallel orthoconvex polygon OP (see Figure 1 for illustration). The problem of locating the largest *type-2* MEC with p_i on its bottom face reduces to finding the largest MER inside OP containing the point p_i. We explain the method of computing $STAIR_1$. The other stairs are computed in a similar manner. Next, we explain the method of computing $MER(p_i)$ in OP.

3.2.1 Computation of $STAIR_1$

We sort the points in P_i^1 in increasing order of their y-coordinates. Now, sweep a line parallel to the x-axis on $H(p_i)$ to identify $STAIR_1$. The points in $STAIR_1$ are maintained at the begining of the array P_i^1, and the points in P_i^1 that are not in $STAIR_1$, are stored at the end of P_i^1. The points in $STAIR_1$ are stored in decreasing order of their x-coordinates. Two index variables α and β are maintained during the execution; α indicates the index of the point in P_i^1 under processing, and β indicates the index of the last point in $STAIR_1$ (i.e., having minimum x-coordinate among the ones identified so far). During the sweep, if $p_\alpha = (x_\alpha, y_\alpha, z_\alpha) \in P_i^1$ satisfies $x_\alpha > x_\beta$, then p_α does not appear on $STAIR_1$. However, if $x_\alpha < x_\beta$, then p_α appears in $STAIR_1$. In such a case, if $\alpha = \beta + 1$, then both α and β are incremented by 1. But, if $\alpha > \beta + 1$, then (i) β is incremented, (ii) $P_i^1[\alpha]$ and $P_i^1[\beta]$ are swapped, and (iii) α is incremented to process the next point of P_i^1.

3.2.2 Computation of $MER(p_i)$

It is easy to observe that, for every MER inside the orthoconvex polygon OP, its north side will contain a point in $STAIR_1 \cup STAIR_2$, and its south side will contain a point $STAIR_3 \cup STAIR_4$. In our algorithm for computing $MER(p_i)$, we will consider each point in $STAIR_1 \cup STAIR_2$, and compute all the MERs with north side passing through it.

The MERs with north side touching a point $p_j \in STAIR_1$ are obtained as follows. We draw the projections q_1 and q_2 of p_j on $STAIR_2$ and $STAIR_4$ respectively as shown in Figure 2. Let q_1 lies on the vertical line passing through $p_\alpha \in STAIR_2$ and q_2 lies on the horizontal line passing through $p_\beta \in STAIR_4$. Thus, p_α satisfies $y(q_1) \in [y(p_{\alpha'}), y(p_\alpha)]$, where p_α and $p_{\alpha'}$ are two consecutive points on $STAIR_2$. Thus, q_1 can be obtained by performing binary search in $STAIR_2$. Similarly, q_2 can be obtained by performing binary search in $STAIR_4$. Now, we compute the projections of q_1 and q_2 on $STAIR_3$. Let these two points be q_3 and q_4 respectively. Now, two situations may arise:

[$\mathbf{y(q_3) \leq y(q_4)}$:] Here only one MER with p_j on its north boundary is possible. Its west and south sides will contain p_α and p_β respectively; its east side will contain a point $p'_j \in STAIR_1$ adjacent to p_j to the right side ($y(p'_j) < y(p_j)$) or a point $q \in STAIR_4$ adjacent to q_2 to the right side ($y(q) > y(q_2)$). See Figure 2(a).

[$\mathbf{y(q_3) > y(q_4)}$:] Here more than one MER with p_j on its north boundary may exist (Figure 2(b)). Let $\mu_1, \mu_2, \ldots, \mu_m$ be the consecutive points in $STAIR_3$ with $y(\mu_1) < y(\mu_2) < \ldots < y(\mu_m)$, and $y(\mu_r) \in [y(q_3), y(q_4)]$ for $r = 1, 2, \ldots, m$. Similarly, $\nu_1, \nu_2, \ldots, \nu_\ell$ are consecutive points in $STAIR_4$ with $y(\nu_1) < y(\nu_2) < \ldots < y(\nu_\ell)$, and $y(\nu_r) \in [y(q_3), y(q_4]$ for $r = 1, 2, \ldots, \ell$. Here, $\ell + m + 1$ MERs are possible with north boundary passing through p_j. Their south boundaries will pass through $q_2, \mu_1, \mu_2, \ldots, \mu_k, \nu_1, \nu_2, \ldots, \nu_\ell$ respectively. The east and west sides of these MERs are uniquely defined, and are obtained by traversing the stairs in four quadrants.

Lemma 4 *The time complexity of computing the largest type-2 MEC is $O(n^2 \log n)$ with $O(1)$ extra space.*

Proof. We prove this lemma by showing that the time complexity of generating all the *type-2* MECs with p_i on its bottom face is $O(\mathcal{C}_i + n \log n)$; $\mathcal{C}_i$ is the number of such MECs present in $\mathcal{R}$. Processing of the point p_i consists of the following three steps:

[**Step 1:**] Partitioning the points above p_i into P_i^θ, for $\theta = 1, 2, 3, 4$. This needs $O(n)$ time in the worst case.

[**Step 2:**] Computing $STAIR_\theta$, $\theta = 1, 2, 3, 4$. This needs $O(n \log n)$ time since a sorting step among the points

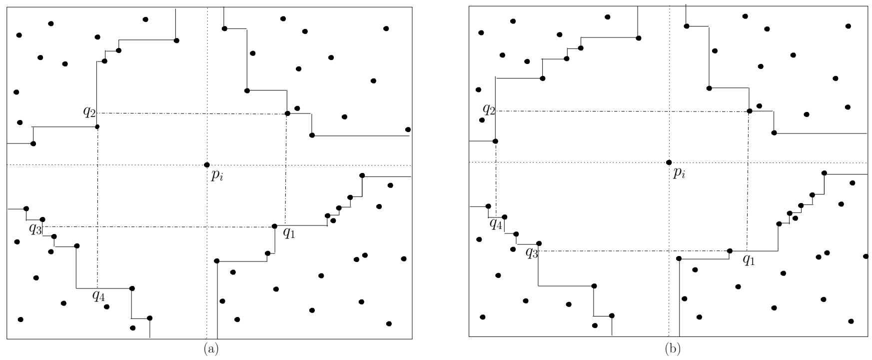

Figure 2: Computation of $MER(p_i)$

in P_i^θ with respect to their y-coordinates is involved here. After the sorting, the line sweep for constructing $STAIR_\theta$ needs $O(n)$ time.

[**Step 3:**] Computing $MER(p_i)$. This needs $O(C_i + n \log n)$ time. The second component in the time complexity appears due to the fact that for each point $p_j \in STAIR_1 \cup STAIR_2$, we need to execute binary searches for computing its projections q_1 and q_2 in the adjacent stairs. Again we may need two binary searches to get the set of feasible points in $STAIR_3$ that may appear in the south boundary of the generated MERs.

Since (i) we need to process all the points $p_i \in P$, (ii) $\mathcal{C} = \sum_{i=1}^{n} \mathcal{C}_i$, and (iii) $|C| = O(n^2)$ in the worst case (see Theorem 2), the time complexity result follows.

We have used four integer locations n_1, n_2, n_3, n_4, six index variables $q_1, q_2, q_3, q_4, \alpha, \beta$, and a space for swap operation. Thus, the space complexity follows. □

3.3 Computation of largest *type-3* MEC

Here we describe the method of generating all the *type-3* MECs with top face passing through a point $p_i \in P$. Let the points in P be in decreasing order of their z-coordinates. Consider the horizontal plane $H(p_i)$ passing through p_i and sweep it downwards. When the sweeping plane hits a point $p_j \in P$, the points inside the two horizontal planes $H(p_i)$ and $H(p_j)$ will participate in computing the MECs with top and bottom faces passing through p_i and p_j respectively.

As in Subsection 3.2.1, here also we use P_i^θ to denote the subset of points in P that lie in θ-th quadrant, $\theta = 1, 2, 3, 4$, determined by the horizontal and vertical lines through the point p_i on $H(p_i)$. The points in $\bigcup_{\theta=1}^{4} P_i^\theta$ are stored in the array-positions $P[i+1], P[i+2], \ldots, P[n]$. The members in P_i^θ are in the consecutive locations of the array P in decreasing order of their z-coordinates. We maintain four integer variables n_θ and four index variables χ_θ, $\theta = 1, 2, 3, 4$. n_θ denotes $|P_i^\theta|$ and χ_θ indicates the last point hit by the sweeping plane in the θ-th quadrant. At an instant of time the point hit by the sweeping plane is obtained by comparing the z-coordinates of the points $\{P[\chi_\theta + 1], \theta = 1, 2, 3, 4\}$.

Let the point p_j be under process. The empty orthoconvex polygon OP around the point p_i is determined by four stairs $\{STAIR_\theta, \theta = 1, 2, 3, 4\}$ using the points lying inside the horizontal slab bounded by $H(p_i)$ and $H(p_j)$ (but not including p_i and p_j). The points determining $STAIR_\theta$ are stored at the begining of the subarray P_i^θ in order of their y-coordinates.

In order to compute the largest MEC with top and bottom faces passing through p_i and p_j respectively, we need to compute $MER(p_i, p_j)$, the largest MER in the orthoonvex polygon OP that contains both p_i and projection p'_j of p_j on $H(p_i)$. Here, the following two tasks need to be performed: (i) Computing all the MERs in OP that contains both p_i and p'_j, and (ii) updating OP by inserting p'_j for processing the next point p_{j+1}.

3.3.1 Computing $MER(p_i, p_j)$

Without loss of generality, assume that p'_j is in the first quadrant. If p'_j is in some other quadrant, the situation is similarly tackled. We now determine the subset of points in $STAIR_1 \cup STAIR_2$ that can appear in the north boundary of an MER containing both p_i and p'_j.

Let $STAIR_1 = \{q_k, k = 1, 2 \ldots, m\} \subseteq P_i^1$, and the points in $Q = \{q_\alpha, q_{\alpha+1}, \ldots, q_\beta\} \subseteq STAIR_1$ satisfy $x(q_k) > x(p_j)$ and $y(q_k) > y(p_j)$. All the MERs in OP with north boundary passing through q_k, $k = \alpha, \alpha + 1, \ldots, \beta + 1$ and containing p_i in its proper interior will contain p'_j also. We draw the projections of p'_j and q_β on $STAIR_2$. Let these two points be μ and ν respectively. If $x(\mu) = x(\nu)$, then no point on $STAIR_2$ can appear on the north boundary of a desired MER. But if $x(\mu) < x(\nu)$, then all the points $q' \in STAIR_2$ satisfying $x(\mu) < x(q') < x(\nu)$ can appear on the north boundary of a desired MER. In Figure 3, the set of points that can appear on the north boundary of an MER are marked with empty dots. The method of computing an MER with a point $q_k \in STAIR_1 \cup STAIR_2$ on its north boundary is same as that in Subsection 3.2.2.

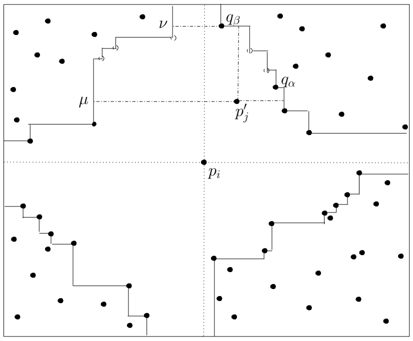

Figure 3: Computation of *type-3* MEC

3.3.2 Updating OP

After computing the set of MERs in OP containing p_i and p'_j in its interior, we update OP by inserting p'_j in the respective stair. We have already assumed that p'_j lies in the first quadrant, and each member $q_k \in Q$ satisfies $x(q_k) > x(p_j)$ and $y(q_k) > y(p_j)$. In order to insert p'_j in $STAIR_1$, we need to remove the members in Q from $STAIR_1$. We maintain two index variables α and β; α indicates the last point of $STAIR_1$ observed so far, and β indicates the point p_j under consideration in P_i^1, $\alpha \leq \beta - 1$. If $\alpha < \beta - 1$, then the points in the positions $\alpha+1, \ldots, \beta-1$ of P_i^1 are already considered, but their projections are not present in $STAIR_1$. While inserting p'_j in $STAIR_1$, we place p_j in its desired location as follows: (i) swap $P[\beta+1]$ and $P[\alpha]$, and then (ii) execute a sequence of swap swap$(P[r], P[r-1])$ starting from $r = \beta + 1$ until a point $P[r] \in STAIR_1$ is found such that $y(P[r]) < y(P[r-1])$. Now, if $|Q| > 0$, then we remove the members in Q using two index variables r and s. We start with $r = \gamma + 1$ and $s = \gamma + |Q| + 1$. At each step, we execute swap$(P[r], P[s])$ and increment r and s by 1 until $s = \beta$. This needs $O(\max(|Q|, (\beta - \gamma)))$ time which may be $O(|P_i^1|)$ in the worst case.

After computing the largest *type-3* MEC with p_i on its top boundary, we need to sort the points again with respect to their z-coordinates. This is required for processing p_{i+1}. Thus we have the following result:

Lemma 5 *The time required for processing* p_i *is* $O(n^2 + C'_i)$ *in the worst case, where* C'_i *is the number of* type-3 *MECs with* p_i *on its top boundary.*

Proof. The time required for computing $MER(p_i, p_j)$ may be $O(|P_{ij}| + C_{ij})$, where P_{ij} denotes the number of points inside the horizontal slab bounded by $H(p_i)$ and $H(p_j)$, and C_{ij} denotes the number of MERs containing both p_i and p'_j inside OP with the projection of points P_{ij} on $H(p_i)$. In order to compute the largest *type-3* MEC with p_i on its top boundary, we need to compute $MER(p_i, p_j)$ for all $j > i$, $C'_i = \sum_{j=i+1}^{n} C_{ij}$, and $\sum_{j=i+1}^{n} |P_{ij}| = O((n-i)^2)$. Finally after the processing of p_i, the sorting step takes $O(n \log n)$ time. □

Theorem 6 *The worst case time complexity of our in-place algorithm for computing the largest MEC is* $O(n^3)$*, and it takes* $O(1)$ *extra space.*

Proof. The time complexity for computing the largest *type-1* MEC is $O(n^2)$ (see Corollary 1.1). Lemma 4 and the fact that the number of *type-2* MECs is $O(n^2)$ in the worst case [12], indicate that the worst case time complexity of computing the largest *type-2* MEC is alo $O(n^2 \log n)$. Finally, Lemma 5 says that the worst case time complexity of computing the largest *type-3* MEC is $O(n^3)$. Needless to mention that we have used only few index variables, four integer variables to maintain the number of points in the four quadrants on $H(p_i)$, and a temporary variable for the swap operation. □

References

[1] A. Aggarwal and S. Suri. Fast algorithm for computing the largest empty rectangle. In *Symp. on Comput. Geom.*, pages 278-290, 1987.

[2] T. Asano and G. Rote. Constant working-space algorithms for geometric problems. In *Canad. Conf. on Comput. Geom.*, pages 87-90, 2009.

[3] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. H. M. Smid and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry*, 37(3):209-227, 2007.

[4] H. Brönnimann, T. M. Chan and E. Y. Chen. Towards in-place geometric algorithms and data structures. In *Symp. on Comput. Geom.*, pages 239-246, 2004.

[5] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison and G. T. Toussaint. Space-efficient planar convex hull algorithms. *Theoretical Computer Science*, 321(1):25-40, 2004.

[6] J. Chaudhuri, S. C. Nandy and S. Das. Largest empty rectangle among a point set. *J. Algorithms*, 46(1):54-78, 2003.

[7] E. Y. Chen and T. M. Chan. A space-efficient algorithm for segment intersection. In *Canad. Conf. on Comput. Geom.*, pages 68-71, 2003.

[8] T. M. Chan, E. Y. Chen. Optimal in-place algorithms for 3-D convex hulls and 2-D segment intersection. In *ACM Symp. on Comput. Geom.*, pages 80-87, 2009.

[9] M. De, A. Maheswari, S. C. Nandy and M.Smid. An in-place min-max priority search tree. *Manuscript*, 2011.

[10] H. Edelsbrunner. Algorithms in Combinatorial Geometry, Springer, Berlin, 1987.

[11] R. Janardan and F. P. Preparata. Widest corridor problem. *Nordic J. Computing*. 1(2):231-245, 1994.

[12] S. C. Nandy and B. B. Bhattacharya. Maximal empty cuboids among points and blocks. *Computers and mathematics with Applications*, 36(3):11-20, 1998.

[13] A. Naamad, D. T. Lee and W. -L. Hsu. On the maximum empty rectangle problem. *Discrete Applied Mathematics*, 8(3):267-277, 1984.

[14] M. Orlowski. A new algorithm for the largest empty rectangle problem. *Algorithmica*, 5(1-4):65-73, 1990.

[15] J. Vahrenhold. An in-place algorithm for Klee's measure problem in two dimensions. *Information Processing Letters*, 102(4):169-174, 2007.