ELSEVIER

SCIENCE DIRECT

Computational Geometry 29 (2004) 163–175

Computational Geometry
Theory and Applications

www.elsevier.com/locate/comgeo

Triangular range counting query in 2D and its application in finding k nearest neighbors of a line segment ☆

Partha P. Goswami [a], Sandip Das [b], Subhas C. Nandy [b,*]

[a] *Computer Center, Calcutta University, Kolkata, 700 029, India*
[b] *ACM Unit, Indian Statistical Institute, Kolkata, 700 108, India*

Received 21 June 2002; received in revised form 21 December 2003; accepted 26 February 2004

Available online 4 May 2004

Communicated by Chee Yap

Abstract

We present an efficient algorithm for finding k nearest neighbors of a query line segment among a set of points distributed arbitrarily on a two dimensional plane. Along the way to finding this algorithm, we have obtained an improved triangular range searching technique in 2D. Given a set of n points, we preprocess them to create a data structure using $O(n^2)$ time and space, such that given a triangular query region Δ, the number of points inside Δ can be reported in $O(\log n)$ time. Finally, this triangular range counting technique is used for finding k nearest neighbors of a query straight line segment in $O(\log^2 n + k)$ time.

Keywords: Triangular range counting; Segment dragging query; Arrangement; Duality

1. Introduction

Given a set $P = \{p_1, p_2, \ldots, p_n\}$ of n points arbitrarily distributed on a 2D plane, we study the problem of finding k nearest neighbors of a query line segment σ. Along the way to studying this problem, we have developed an improved algorithm for the *counting query of triangular ranges* in 2D, where the objective is to report the number of points in P that lie inside a query triangle.

A simplex in $\mathcal{R}^d$ is a space bounded by $d + 1$ hyperplanes. In the simplex range query problem, a set of points P (in $\mathcal{R}^d$) is given; the objective is to report the number/subset of points which lie inside a

☆ A preliminary version of this paper was presented in SWAT 2002.

* Corresponding author.
E-mail address: nandysc@isical.ac.in (S.C. Nandy).

0925-7721/$ – see front matter
doi:10.1016/j.comgeo.2004.02.001

simplex query region. We shall refer these two problems as *counting query problem* and *subset reporting problem* respectively. The simplex range searching problem has been studied extensively [3,9,11,15]. The first work appeared in [11], where two near quadratic data structures are presented; the first one can answer triangular range counting query in the plane in $O(\log n)$ time using $O(n^{2+\varepsilon})$ space, and the second one needs $O(\log n \log\log n)$ time for the counting query using $O(n^2/\log n)$ space. In $\mathcal{R}^3$, this algorithm answers the simplex range counting query in $O(\log n)$ time, but it requires $O(n^{7+\varepsilon})$ space for storing the preprocessed data structure, where ε is a fixed positive constant. Chazelle, Sharir and Welzl [9] presented a quasi-optimal upper bound for the triangular range searching problem. Their algorithm can achieve $O(\log n)$ time for the *counting query* in $\mathcal{R}^2$ at the expense of $O(n^{2+\varepsilon})$ storage and preprocessing time. The best result is obtained by Matousek [15]. In $\mathcal{R}^2$, it needs $O(n^2 \log^{\varepsilon} n)$ preprocessing time and $O(n^2)$ space. Using a weight function of the points, *counting query* can be answered in $O(\log^3 n)$ time. For both of these algorithms, the subset reporting problem needs an additional $O(\kappa)$ time, where κ is the size of the output. For a concise survey on geometric range searching, see [2]. In this paper, we present an improved algorithm which solves the *counting query* problem in $O(\log n)$ time with $O(n^2)$ preprocessing time and space. However, this technique does not help in solving the *subset reporting query* problem. We mention a different technique which needs $O(n^2)$ preprocessing time and space, and answers the *subset reporting query* in $O(\log^2 n + \kappa)$ time.

The problem of computing the nearest neighbor of a query line was initially addressed in [11]. An algorithm of preprocessing time and space $O(n^2)$ was proposed in that paper which can answer the query in $O(\log n)$ time. Later, the same problem was solved using geometric duality in [14], with the same time and space complexities. The space complexity of the problem has recently been improved in [16,17]. In [16], the preprocessing time and space are reduced to $O(n \log n)$ and $O(n)$ respectively, and the query time is $O(n^{.695})$. The query time is further reduced to $O(n^{\frac{1}{2}+\varepsilon})$, but this needs $O(n \log n)$ space to store the preprocessed data structure [17]. In [18], the problem of finding the k nearest neighbors of a line is proposed along with an algorithm. The preprocessing time and space complexities are $O(n^2)$ and $O(n^2/\log n)$ respectively, and the query time complexity is $O(k + \log n)$, where k is an input at the query time. The same data structure can report k farthest neighbors of a query line with same time and space complexities. The proximity queries regarding line segments are studied very little. The first work appeared in [5] which addresses the problem of locating the nearest point of a query straight line segment among a set of n points on the plane. The results obtained in few restricted cases of this problem are listed below.

(i) If the query line segment is known to lie outside the convex hull of P, a linear size data structure can be constructed in $O(n \log n)$ time, which can answer the nearest neighbor of a line segment in $O(\log n)$ time.
(ii) If m non-intersecting query line segments are given at a time, then the nearest neighbors of all these line segments can be reported in $O(m \log^3 n + n \log^2 n + m \log m)$ time.
(iii) When n arbitrary query line segments (they may intersect) are given in advance, then the nearest neighbors of all of them can be reported in $O(n^{4/3} \log^k n)$ time, for some constant k.
(iv) If n query line segments arrive online, then the amortized query time per segment is $O(\sqrt{n} \log^k n)$, for some constant k.

The time complexity of problems (ii) and (iii) are improved in [4]. If the number of points and query line segments are both n, the time complexities are $O(n \log^2 n)$ and $O(n^{4/3} 2^{O(\log^* n)})$ respectively.

We consider an unrestricted version of the nearest neighbor query problem for line segments. The objective is to report the nearest neighbor of an arbitrary query line segment σ among a set of points P. We show that the preprocessed data structure for the simplex range searching problem can answer this query in $O(\log^2 n)$ time. We also show that the following queries can be answered using our method:

Segment dragging query: Report the first k points (of P) hit by the query line segment σ if σ is dragged along any one of its two perpendicular directions. This needs $O(k + \log^2 n)$ time. The preprocessing time and space complexities are both $O(n^2)$.

k-nearest neighbors query: This problem has two phases: (i) find k nearest neighbors of the interior of σ, and (ii) find k nearest neighbors of each end point of σ. The first phase can be solved using *segment dragging query* technique. The second phase needs to use order-k Voronoi diagram, provided k is known prior to the preprocessing. Thus the preprocessing and query time complexities of the order-k Voronoi diagram play important role in analyzing the complexity results of this problem.

2. Preliminaries

Let Q be a set of n points distributed arbitrarily on a 2D plane. First, we describe a method of *counting* and *subset reporting* for the *half-plane range query* for a query line ℓ using geometric duality. Here (i) a point $q = (a, b)$ of the primal plane is mapped to the line q^*: $y = ax - b$ in the dual plane, and (ii) a non-vertical line ℓ: $y = cx - d$ of the primal plane is mapped to the point $\ell^* = (c, d)$ in the dual plane. A point q is below (respectively, on, above) a line ℓ in the primal plane if and only if the line q^* is above (respectively, on, below) the point ℓ^* in the dual plane.

Let Q^* be the set of dual lines corresponding to the points in Q, and $\mathcal{A}(Q^*)$ denotes the arrangement of the lines in Q^*. As a preprocessing, we construct a data structure for storing the levels of the arrangement $\mathcal{A}(Q^*)$ as defined below. From now onwards, this data structure will be referred to as *level-structure*.

Definition 1 [12]. A point q in the dual plane is at level θ $(0 \leqslant \theta \leqslant n)$ if there are exactly θ lines in Q^* that lie strictly below q. The θ-level of $\mathcal{A}(Q^*)$ is the closure of a set of points on the lines of Q^* whose levels are exactly θ in $\mathcal{A}(Q^*)$, and is denoted as λ_θ.

Clearly, the edges of λ_θ form a monotone polychain from $x = -\infty$ to $x = \infty$. Each vertex of the arrangement $\mathcal{A}(Q^*)$ appears in two consecutive levels, and each edge of $\mathcal{A}(Q^*)$ appears in exactly one level. In Fig. 1, a demonstration of levels in an arrangement of lines is shown. The thick chain

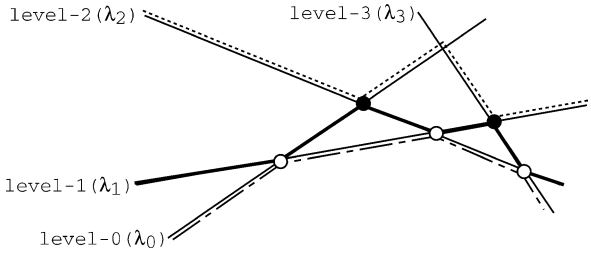

Fig. 1. Demonstration of levels in an arrangement of lines.

represents λ_1. Among the vertices of λ_1, those marked with empty (respectively black) circles are appearing in level λ_0 (respectively λ_2) also.

Definition 2. The *level-structure* is an array $\mathcal{A}$ whose elements correspond to the levels $\{\theta \mid \theta = 1, \ldots, n\}$ of the arrangement $\mathcal{A}(Q^*)$. Each element representing a level θ, is attached with a linear array containing the vertices of λ_θ in a left to right order.

Given a query line ℓ, the *counting query* problem for the halfplane range is solved as follows: Find two consecutive levels θ and $\theta + 1$ such that ℓ^* (dual of the line ℓ) lies within λ_θ and $\lambda_{\theta+1}$. This can be done by two stage binary search in the level-structure.

First we select the $\lceil \frac{n}{2} \rceil$th level in the array $\mathcal{A}$, and apply binary search in $\lambda_{\lceil \frac{n}{2} \rceil}$ to find an edge $e \in \lambda_{\lceil \frac{n}{2} \rceil}$ which horizontally spans ℓ^*. Next we choose either $\lceil \frac{n}{4} \rceil$th level or $\lceil \frac{3n}{4} \rceil$th level depending on whether ℓ^* appears below or above e. The process continues until we get a pair of edges e^* and e^{**} below and above ℓ^* respectively such that they appear in two consecutive levels of the arrangement, i.e., $e^* \in \lambda_{\theta-1}$ and $e^{**} \in \lambda_\theta$. The number of points above and below the query line ℓ are θ and $n - \theta$ respectively. The time complexity of this half-plane range counting query algorithm is $O(\log^2 n)$.

To improve the query time, an augmentation procedure of the *level-structure* is described in [18]. This uses a technique similar to *fractional cascading* [8]. The complexity results of half-plane range query using the *augmented level-structure* is stated in the following theorem.

Theorem 1 [18]. *Given a set of n points in the plane, it can be preprocessed in $O(n^2)$ time and $O(n^2)$ space such that the counting query for the half-plane range can be answered in $O(\log n)$ time. The subset reporting needs an additional $O(\kappa)$ time, where κ is the size of the output.*

3. Triangular range searching

Given a triangular range Δ, here the objective is to report the points in P that lie inside Δ. We shall consider both *counting* and *subset reporting* query problems for the triangular range separately.

3.1. Preprocessing

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n points arbitrarily distributed in 2D plane. A line splitting the set P into two non-empty subsets is called a *cut*. A cut is said to be *balanced* if it splits P into two subsets P_1 and P_2 such that the size of these two subsets differ by at most one. For a given point set P, the balanced cut may not be unique. But we may choose any one of them. The point sets P_1 and P_2 are further divided recursively using balanced cuts. This process continues until all the partitions contain exactly one element.

As a preprocessing for the triangular range query, we create a data structure $T(P)$, called *partition tree*, based on the hierarchical balanced bipartitioning of the point set P. Its root node (at layer-0) corresponds to the set P. The cut I_0 splits P into two subsets P_1 and P_2 and they correspond to the left and right child of the root (i.e., two nodes at layer-1). The points in P_1 and P_2 are further partitioned using balanced cuts to create four nodes at layer-2. The splitting continues and the nodes of the tree are defined recursively in this manner.

Each node v of $T(P)$ is attached with (i) the set of points P_v, (ii) an integer field χ_v indicating the size of P_v, and (iii) the balanced cut line I_v partitioning the point set P_v.

In order to define the partition tree $T(P)$ uniquely, we use *ham-sandwich cuts* (defined below) as balanced cuts for the point sets attached to the pair of children of each non-leaf node.

Definition 3. Let Q be a set of points and I be a balanced cut for Q which creates two subsets Q_1 and Q_2. A *ham-sandwitch* cut of Q_1 and Q_2 is a line J which splits both Q_1 and Q_2 into two equal halves.

Lemma 1 [13]. *Given a set P of n points, the partition tree $T(P)$ can be constructed in* $O(n \log n)$ *time and space.*

After obtaining the partition tree, we attach two secondary structures, namely SS_1 and SS_2, with each non-leaf node of $T(P)$. SS_1 is a *level structure* with the points in P_v. It is useful for the triangular range counting query, but is not efficient for the subset reporting. SS_2 is another data structure with the same point set P_v, and is used for reporting the members inside the query triangle. The performance of SS_2 with respect to the counting query is inferior to that of SS_1.

3.1.1. Secondary structure—SS_1

Consider the node v in $T(P)$. P_v^* is the set of lines corresponding to the duals of the points in P_v. We create the augmented level structure $\mathcal{A}(P_v^*)$ (as mentioned in Section 2), and attach it with node v. From now onwards, this secondary structure will be referred as $SS_1(v)$. We further augment the data structure using the following lemma. This accelerates the triangular range counting query.

Lemma 2. *Let v be a node in $T(P)$, and w be a child node of v. A cell in $\mathcal{A}(P_v^*)$ is completely contained in exactly one cell of $\mathcal{A}(P_w^*)$.*

Proof. Let u be the other child of node v. P_u^* and P_w^* are the duals of the points attached to nodes u and w. Let C be a cell in $\mathcal{A}(P_v^*)$. It is bounded by the lines of both P_u^* and P_w^*. If the lines of P_u^* are removed, C will be contained in a cell of the arrangement $\mathcal{A}(P_w^*)$. □

With each cell $\mathcal{C} \in \mathcal{A}(P_v^*)$, we attach two pointers, $cell\_ptr_{left}$ and $cell\_ptr_{right}$. They point to the cells $\mathcal{C}_L \in \mathcal{A}(P_u^*)$ and $\mathcal{C}_R \in \mathcal{A}(P_w^*)$ respectively in which the cell $\mathcal{C}$ is contained (we have assumed that u and w are respectively the left and right child of node v in $T(P)$). Basically, this can be done by attaching the pointers with each edge $e \in \mathcal{A}(P_v^*)$ as follows:

- If the edge e is a part of an edge $e^* \in \mathcal{C}_L$, then its $cell\_ptr_{left}$ points to e^*. We draw a vertical line at the left end point of e in downward direction. Let it hit the edge $e^{**} \in \mathcal{C}_R$. The $cell\_ptr_{right}$ points to e^{**}.
- If e is a part of an edge in $\mathcal{C}_R$, then its $cell\_ptr_{right}$ and $cell\_ptr_{left}$ are computed in a similar manner.

Lemma 3. *The time and space required for creating and storing the preprocessed data structure SS_1 for all non-leaf nodes in $T(P)$ are both* $O(n^2)$.

Proof. By Lemma 1, the initial partition tree can be constructed in $O(n \log n)$ time and space. The number of nodes at level i of $T(P)$ is 2^i, and each of them contains at most $\lceil \frac{n}{2^i} \rceil$ points, where

$i = 0, 1, \ldots, \log n - 1$. For each non-leaf node at level i, the size of SS_1 structure is $O((\frac{n}{2^i})^2)$, and it can be constructed from the point set assigned to that node in $O((\frac{n}{2^i})^2)$ time. So, the total time and space required for constructing the SS_1 structure for all nodes in $T(P)$ is

$$O\left(\sum_{i=0}^{\log n-1} 2^i \times \left(\frac{n}{2^i}\right)^2\right) = O\left(\sum_{i=0}^{\log n-1} \frac{n^2}{2^i}\right) = O(n^2).$$

Finally, we use topological line sweep to set $cell\_ptr_{left}$ and $cell\_ptr_{right}$ attached to each edge of $\mathcal{A}(P_v^*)$. This requires an additional $O(n^2)$ amount of time. □

3.1.2. Secondary structure—SS_2

This is another secondary structure attached to each non-leaf node v of $T(P)$, and is created with the points attached to node v in the primal plane.

Consider a non-root node v at ith layer of the tree $T(P)$. The region of the plane attached to it is R_v, and the set of points attached to it is P_v. Note that, R_v is a convex polygonal region whose boundaries are defined by the cut lines of its predecessors, i.e., all the nodes on the path from the root of $T(P)$ up to the parent of the current node. Thus, if v is at layer-i, the number of sides of the region R_v is at most i. We store the boundary edges of R_v in an array. Each edge I is attached with the following data structure.

Let π be a point on an edge I of the boundary of R_v. A half-line is rotated around π inside the region R_v by an amount 180°, and a list L_π is formed with the points in P_v ordered with respect to their appearance during the rotation. For each point of I we get such a list. Note that, we may get an interval on I around the point π such that for all points inside this interval, the list remains same. In order to get these intervals, we join each pair of points in P_v by a straight line. These lines are extended in both sides up to the boundary of R_v. This creates at most $O(|P_v|^2)$ intervals along the boundary of R_v. If we consider any two consecutive intervals on an edge I of the boundary of R_v, the circular order of points only differ by a swap of two members in their respective lists. This indicates that $O(|P_v|^2)$ space is enough to store the circular order of the points of P_v for all the intervals on I. Indeed, we can use the data structure proposed in [10] for storing almost similar lists for this purpose. For the details about this data structure, see [10,11].

Lemma 4. *The time and space required for creating and storing the SS_2 data structure for all nodes in $T(P)$ is $O(n^2)$.*

Proof. Each point appears in exactly one region in each layer of $T(P)$. But a point p inside a region R_v (corresponding to a node v) appears in the data structure attached to all the edges of R_v. For a node v at the ith layer of $T(P)$, its attached R_v is bounded by at most i cut-lines, and contains $\frac{n}{2^i}$ points. Thus the total space required to store the data structure for all edges on the boundary of R_v is at most $i \times (\frac{n}{2^i})^2$. Again, the number of nodes in the ith layer is 2^i. Thus, the time and space required for creating and storing the SS_2 data structure for all nodes in $T(P)$ in the worst case is equal to

$$\begin{aligned} &1.2^1.\left(\frac{n}{2}\right)^2 + 2.2^2.\left(\frac{n}{2^2}\right)^2 + 3.2^3.\left(\frac{n}{2^3}\right)^2 + \cdots + \log n.2^{\log n}.\left(\frac{n}{2^{\log n}}\right)^2 \\ &= \frac{1}{2}.n^2 + \frac{2}{2^2}.n^2 + \frac{3}{2^3}.n^2 + \cdots + \frac{\log n}{2^{\log n}}.n^2 \\ &= O(n^2). \qquad \square \end{aligned}$$

3.2. Counting query

Here the objective is to report the number of points of P that lie inside a triangular query region Δ. We traverse the preprocessed data structure $T(P)$ from its root with the region Δ. A global *COUNT* field (initialized with 0) is maintained during the traversal. At the end of traversal, *COUNT* field returns the number of points inside Δ.

During the traversal, if a leaf node is reached with a query region $\Delta^*(\in \Delta)$, the *COUNT* is incremented by one if the point attached to that node lies inside Δ^*. While processing a non-leaf node v with a query region Δ^*, its attached partition line I_v may or may not split Δ^*. In the former case, v is said to be a *split node*, and in the latter case v is said to be a *non-split node*.

At a *non-split node* v, the traversal proceeds towards one child of v whose corresponding partition contains Δ^*. On the other hand, at a *split node*, Δ^* splits into two query regions; each of them is convex, and is any one of the following types:

type-0: a region having no corner of Δ,
type-1: a region having one corner of Δ, and
type-2: a region having two corners of Δ.

When Δ splits for the first time, it gives birth to one *type*-1 and one *type*-2 regions. In the successive splits:

- A *type*-2 region may either be split into (i) one *type*-0 region and one *type*-2 region, or (ii) two *type*-1 regions. In case (i), the *counting query* inside the *type*-0 region is performed among the points in the partition attached to one child of node v which contains the *type*-0 region; traversal proceeds towards the other child of v containing the *type*-2 region. In case (ii), traversal proceeds towards both the children of v with the corresponding *type*-1 region, in a recursive manner.
- A *type*-1 region splits into one *type*-0 and one *type*-1 region. The processing of *type*-0 region is performed at one child of node v, and the traversal proceeds towards the other child of v with the *type*-1 region.

The processing of a *type*-0 region at a node v is described below.

Lemma 5. *During the entire traversal with the query region Δ, the number of type*-0 *regions that may be generated is at most* $O(\log n)$.

Proof. After the split of Δ into a *type*-2 and a *type*-1 region, we first consider the processing of the *type*-2 region. It may generate $O(\log n)$ *type*-0 regions before it splits into two *type*-1 regions or reaches to the leaf node. Thus, we may have at most three *type*-1 regions.

During the traversal in $T(P)$ with a *type*-1 region, it may split at most $O(\log n)$ times before the search reaches a leaf node. At each of these splits a new *type*-0 region is generated. □

3.2.1. Counting query inside a type-0 region

Lemma 6. *The number of edges of the query triangle Δ that can appear on the boundary of a type*-0 *region is at most three.*

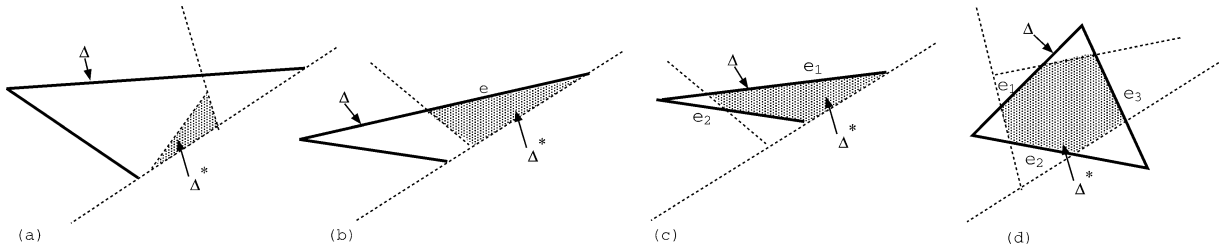

Fig. 2. Different possibilities of *type*-0 region.

Proof. Follows from the fact that at most one segment of a straight line can appear as an edge on the boundary of a convex region, and Δ is bounded by three straight line segments. □

Fig. 2 demonstrates the implication of Lemma 6. Here the boundary of the query triangle Δ is shown with solid lines. The hierarchical partition lines corresponding to the nodes of the partition tree $T(P)$ are shown with dotted lines.

Let Δ^* be a *type*-0 region generated at node v during the traversal of $T(P)$ with the query region Δ. In order to solve the *counting query* with respect to Δ^* at node v, we need to consider the following four cases depending on the number of sides of Δ that appear on the boundary of Δ^*.

Case 1: Δ^* is not bounded by any of the edges of Δ (see Fig. 2(a)). Here all the $\chi_v = |P_v|$ points lie inside Δ^*.

Case 2: Exactly one edge e of Δ appears on the boundary of Δ^* (see Fig. 2(b)). The edge e cuts the boundary of R_v (region attached to node v) in exactly two points, and it splits the point set P_v into two disjoint subsets. One of these subsets lies completely outside Δ^*, and the other one completely lies inside Δ^*. The number of points of P_v lying inside Δ^* can be obtained by performing *half-plane range counting query* (see Section 2) among the point set P_v with respect to the line containing e.

Case 3: Exactly two edges of Δ appear on the boundary of Δ^* (see Fig. 2(c)). These two edges are mutually non-intersecting inside R_v, and each of them intersects the boundary of R_v in exactly two points. Let these two edges be e_1 and e_2. As stated earlier, each of these edges (e_i) partitions the point set P_v into two disjoint subsets, say $P_v(e_i)$ and $\overline{P_v(e_i)}$. $\overline{P_v(e_i)}$ lies completely outside Δ^*, but $P_v(e_i)$ may not completely lie inside Δ^* due to the constraints imposed by the other member e_j, $j \neq i$.

The number of points of P_v inside Δ^* is equal to $|P_v(e_1) \cap P_v(e_2)| = (\chi_v - |\overline{P_v(e_1)}| - |\overline{P_v(e_2)}|)$, and it can be obtained by performing the *half-plane range counting query* with the lines e_1 and e_2 separately.

Case 4: Exactly three edges of Δ appear on the boundary of Δ^* (see Fig. 2(d)). As in case 3, all these edges are mutually non-intersecting inside R_v, and each of them intersects R_v in exactly two points. The number of points of P_v inside Δ^* is equal to

$$\left| \bigcap_{i=1}^{3} P_v(e_i) \right| = \left(\chi_v - \sum_{i=1}^{3} |\overline{P_v(e_i)}| \right),$$

where $P_v(e_i)$ is defined as in case 3. Thus, we have the following lemma.

Lemma 7. *The number of points inside a* type-0 *region can be obtained in* $O(\log n)$ *time in the worst case.*

Proof. From the above discussions, it is clear that after reaching node v with a *type*-0 region Δ^*, the number of points inside Δ^* can be obtained by applying the *half-plane range counting query* at most three times (with respect to the lines containing e_1, e_2 and e_3) among the points in P_v. This requires $O(\log n)$ time in the worst case (see Theorem 1). □

3.2.2. Time complexity of counting query

Lemmata 5 and 7 say that given a triangular query region Δ, the number of points inside it can be reported in $O(\log^2 n)$ time. We now show that the secondary structure SS_1 (augmented using Lemma 2) helps in reducing the counting query cost to $O(\log n)$.

The triangular range counting query starts from the root of $T(P)$ and with *COUNT* equal to zero. Let ℓ_1^*, ℓ_2^* and ℓ_3^* be the dual points of the lines containing the three edges e_1, e_2 and e_3 of Δ. We find the cells containing ℓ_1^*, ℓ_2^* and ℓ_3^* in the SS_1 structure attached to the root node of $T(P)$. During the traversal with a point (say ℓ_1^* or ℓ_2^* or ℓ_3^*), when the search moves from a node v to its children, the cell corresponding to that point in the secondary structure of the children of v are reached using $cell\_ptr_{left}$ or $cell\_ptr_{right}$ in $O(1)$ time. If a *type*-0 region Δ^* is generated at a non-leaf node on the traversal path, *COUNT* is incremented by the number of points inside Δ^*, which is computed as described in cases 1 to 4. This needs $O(1)$ time because of the following two reasons:

- The number of edges of Δ^* crossing the region R_v attached to v is at most 3 (see Lemma 6).
- For each of these edges, the number of points of P_v lying outside Δ^* can be obtained by observing the level of the dual point of the corresponding line in the arrangement $\mathcal{A}(P_v^*)$. This requires $O(1)$ time since the dual point is already placed in the appropriate cell of the arrangement $\mathcal{A}(P_v^*)$.

If a leaf node is reached, the point attached with it is tested to check whether it lies inside Δ in $O(1)$ time. If so, *COUNT* field is incremented by 1. Finally, at the end of traversal, *COUNT* field indicates the number of points inside Δ. Lemma 5 and the aforesaid discussion lead to the following theorem:

Theorem 2. *Given a set of n points we can preprocess them in $O(n^2)$ time and space such that the number of points inside a query triangle can be obtained in $O(\log n)$ time.*

Here, we need to note the following two important things:

- The drawback of using SS_1 secondary structure is that it is not efficient for reporting the set of points inside Δ. Here, the worst case time required for the subset reporting query may be $O(\kappa \log n)$, where κ is the number of points inside Δ.
- One can easily create a *level-structure* using $O(n^2/\log n)$ space and $O(n^2)$ time, which can determine the level of a point (dual of a line) in $O(\log n)$ time [7,18]. But here we cannot use this data structure due to the following reason:

  In order to solve the counting query, we need to solve the point location query (for the dual (point) of a line ℓ) in the SS_1 structure at each level of the partition tree. For a particular point, say ℓ^*, we can reach its corresponding cell at a particular level of the partition tree from that of its predecessor node using either $cell\_ptr_{left}$ or $cell\_ptr_{right}$ in $O(1)$ time by virtue of Lemma 2. If we use $\frac{1}{\log n}$-cutting tree, Lemma 2 will not remain valid. Thus, for each line bounding the query triangle, we need to solve the point location query (for the

dual of that line) at each level of the partition tree in $O(\log n)$ time. Thus, the overall time complexity of the counting query will increase to $O(\log^2 n)$ in the worst case.

3.3. Subset reporting query

The subset reporting for a triangular query region Δ is also done by traversing $T(P)$. At each split node, if the query region is either *type*-1 or *type*-2, it splits in a similar manner as described in the counting query. While processing a *type*-0 region at a particular node, either of the four cases, as mentioned in Section 3.2.1, may appear. The processing of those cases are described below.

- In case 1, all the points in P_v are reported.
- In case 2, the subset of points of P_v lying in one side of the edge e (of Δ) are reported. This can be easily done using SS_1 data structure itself.
- In case 3, the query region at node v is bounded by two edges, say e_1 and e_2, of Δ. We use the secondary structure SS_2 for the reporting in this case. The edges e_1 and e_2 are indicated by bold lines in Fig. 3. Edge e_1 (respectively e_2) intersects R_v at α_1 and α_2 (respectively β_1 and β_2). We split the query region by the diagonal $\alpha_1\beta_2$ indicated by dotted line in Fig. 3. The points α_1 and β_2 lies on the edges I and J of R_v respectively. We use binary search with the point α_1 (respectively β_2) to locate its corresponding interval on the edge I (respectively J). Next we use the list attached to I (respectively J) in SS_2 for reporting the points inside the lightly (darkly) shaded angular region. The detailed method is described in [10]. The time complexity of this reporting is $(\log n + \kappa)$, where κ is the size of the output.
- In case 4, the query region at node v is bounded by three edges e_1, e_2 and e_3 of Δ. Here we need to proceed the traversal to both the children of v in $T(P)$. If the query region splits at v, it may generate at most one query region which is bounded by three edges of Δ. Traversal proceeds with this part. The other part of the split is bounded by either one or two edges of Δ. The points inside this part are reported as in case 1 or case 2 or case 3. This type of split takes place at most $(\log n)$ time. So the reporting time in this case may be $O(\log^2 n + \kappa)$ in the worst case.

Theorem 3. *Given a set of n points we can preprocess them in $O(n^2)$ time and space such that the subset of points inside a query triangle can be reported in $O(\log^2 n + k)$ time.*

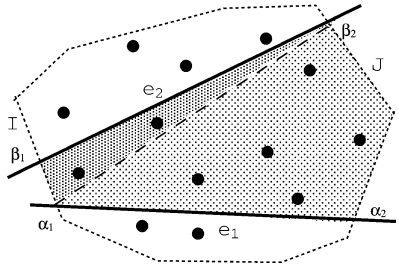

Fig. 3. Reporting in case 3.

Proof. Due to the successive splitting during the traversal in $T(P)$ with the query region Δ, at most $O(\log n)$ *type*-0 query regions are generated (see Lemma 5) for which the subset reporting needs to be done. At most one of these *type*-0 query regions may satisfy case 4 (see Fig. 2(d)). Apart from reporting the output, the search time for this *type*-0 region (if exists) needs $O(\log^2 n)$ time. For all other *type*-0 regions, the total search time is $O(\log^2 n)$. Thus, the time complexity of the subset reporting for the triangular range is $O(\log^2 n + \kappa)$, where κ is the number of reported answers. □

4. Segment dragging query

We shall use the preprocessed data structure discussed in the earlier section for solving the segment dragging query with respect to a line segment $\sigma = [\alpha, \beta]$, where α and β are two end-points of σ.

Let us consider a corridor $\mathcal{C}_\sigma$ defined by two parallel lines L_1 and L_2 drawn through the points α and β respectively, and each of them is perpendicular to σ. The set of points inside the corridor is split into two subsets P_{above} and P_{below} by the segment σ. In the segment dragging query, we need to report k nearest points of σ among the members in P_{above} (respectively P_{below}). Here k may be specified at the query time.

Consider the levels of arrangement $\mathcal{A}(P^*)$. Let ℓ_σ denotes the line containing the segment σ. Let ℓ^*_σ (the dual of ℓ_σ) lies between levels λ and $\lambda + 1$ in the dual plane. If $\lambda < k$, then σ hits no more than $\lambda (< k)$ points if it is dragged above up to infinity. So, all the points in P_{above} need to be reported. If $\lambda > k$, we need to find a line segment $\hat{\sigma}$ parallel to σ and touching the two boundaries of the corridor $\mathcal{C}_\sigma$ such that the number of points inside the region R defined by L_1, L_2, σ and $\hat{\sigma}$ (as shown in Fig. 4) is equal to k (excepting the degenerate cases). Thus, here the segment dragging query consists of two phases: (i) compute $\hat{\sigma}$ appropriately, and (ii) report the points inside R.

We solve the first part as follows: draw a vertical ray from the point ℓ^*_σ downwards. Let $e \in \mathcal{A}(P^*)$ be an edge at level θ ($\theta < \lambda - k$) whom the ray hits. Let p^* ($p \in P$) be the line containing the edge e. We draw a line parallel to σ at the point p. This defines a rectangle R_θ bounded by two boundaries of $\mathcal{C}_\sigma$, the given query segment σ, and the portion of the line p^* inside the corridor $\mathcal{C}_\sigma$. Let κ_θ denotes the number of points inside R_θ. We compute κ_θ by splitting R_θ into two triangles, and then applying the triangular range counting method as described in Section 3.

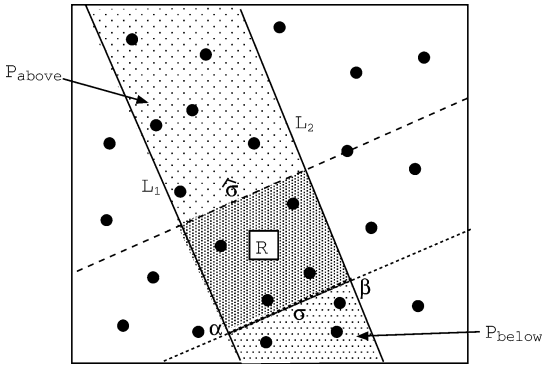

Fig. 4. Segment dragging query for $k = 4$.

Lemma 8. *Let e_i and e_j be two edges at level i and j, $i < j$. R_i and R_j denote two rectangles as defined above. If κ_i and κ_j denote the number of points inside R_i and R_j respectively, then $\kappa_i > \kappa_j$.*

Proof. Follows from the fact that the area of R_i is greater than that of R_j, but the three sides of R_i are common to those of R_j. □

Lemma 9. *A rectangle R above the query line segment σ and containing exactly k points can be obtained in $O(\log^2 n)$ time.*

Proof. We consider all the edges of $\mathcal{A}(P^*)$ that are hit by the vertically downward ray shot from the point ℓ_σ^*. These edges are at different levels of $\mathcal{A}(P^*)$. By Lemma 8, the desired $\hat{\sigma}$ can be selected using binary search among these edges. The lemma follows from the fact that, at each step (i) we choose an edge from the aforesaid set and define the corresponding rectangle inside the corridor C_σ, and then (ii) we find the number of points inside that rectangle by triangular range counting query. □

Next, we use subset reporting algorithm for triangular range to report the points inside the desired rectangle R. Thus, we have the main result of this work.

Theorem 4. *Given a set of n points in 2D, it can be preprocessed in $O(n^2)$ time and space such that for an arbitrary query line segment, the segment dragging query can be answered in $O(k + \log^2 n)$ time, where k is an input at the query time.*

5. k nearest neighbors of σ

The problem of finding k nearest neighbors of $\sigma = [\alpha, \beta]$ has two phases: (i) solve segment dragging query with parameter k for both above and below σ, and (ii) find k nearest neighbors of α and β. Phase (i) outputs two lists L_{above} and L_{below} each containing at most k points. The members of these lists are sorted with respect to their vertical distances from σ. Phase (ii) can be solved by creating the order-k Voronoi diagram provided k is known prior to the preprocessing. This produces two lists L_α and L_β each containing k points. The members in these lists are sorted with respect to their distances from α and β respectively. Finally, a merge like pass among the members in these four lists returns the k nearest neighbors of the line segment σ. Thus, the time complexity of creating an order-k Voronoi diagram influences the preprocessing time complexity for finding k nearest neighbors query of a line segment. The time complexity of the best known deterministic algorithm for creating an order-k Voronoi diagram is $O(nk \log^2 k (\frac{\log n}{\log k})^{O(1)})$ [6]. The storage and query time complexities for the k nearest neighbors problem remain same as that of the segment dragging query problem. It needs to be mentioned that Agarwal et al. [1] proposed a randomized algorithm for creating an order-k Voronoi diagram in $O(n \log^3 n + k(n-k))$ time. Thus, we have the following theorem:

Theorem 5. *Given a set of n points in 2D, it can be preprocessed in $O(n^2)$ time and space such that for an arbitrary query line segment σ, the k nearest neighbors of σ can be computed in $O(k + \log^2 n)$ time, where k is specified at the time of preprocessing.*

Final remark. In case of finding the nearest neighbor of a query line segment σ (i.e., when $k = 1$), the preprocessing time and space complexities are both $O(n^2)$, and query can be answered in $O(\log^2 n)$ time. This is a generalized and improved result of the problem presented in [5]. Note that, here SS_2 data structure need not be maintained with each node of $T(P)$, since the reporting of the segment dragging query can be made using SS_1 data structure.

References

[1] P.K. Agarwal, M. de Berg, J. Matousek, O. Schwartzkopf, Constructing levels in arrangement and higher order Voronoi diagram, SIAM J. Comput. 27 (1998) 654–667.

[2] P.K. Agarwal, J. Erickson, Geometric range searching and its relatives, Advances in Discrete and Computational Geometry, in: B. Chazelle, J.E. Goodman, R. Pollack (Eds.), Proc. of the 1996 AMS-IMS-SIAM Joint Summer Research Conference on Discrete and Computational Geometry: Ten Years Latter, in: Contemporary Mathematics, vol. 223, 1996, pp. 1–56.

[3] P.K. Agarwal, J. Matousek, Dynamic half-space range reporting and its applications, Algorithmica 13 (1995) 325–345.

[4] S. Bespamyatnikh, Computing closest points for segments, Internat. J. Comput. Geom. Appl. 13 (5) (2003) 419–438.

[5] S. Bespamyatnikh, J. Snoeyink, Queries with segments in Voronoi diagram, Computational Geometry 16 (2000) 23–33.

[6] T.M. Chan, Random sampling, halfspace range reporting, and construction of $(\leqslant k)$-levels in three dimension, SIAM J. Comput. 30 (2000) 561–575.

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

[8] B. Chazelle, L.J. Guibas, Fractional cascading—II. Applications, Algorithmica 1 (1986) 163–191.

[9] B. Chazelle, M. Sharir, E. Welzl, Quasi-optimal upper bounds for simplex range searching and new zone theorems, Algorithmica 8 (1992) 407–429.

[10] R. Cole, Searching and storing similar lists, J. Algorithms 7 (1986) 202–230.

[11] R. Cole, C.K. Yap, Geometric retrieval problems, Inform. and Control 63 (1984) 39–57.

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

[13] H. Edelsbrunner, E. Welzl, Halfplanar range search in linear space and $O(n^{.695})$ query time, Inform. Process. Lett. 23 (1986) 289–293.

[14] D.T. Lee, Y.T. Ching, The power of geometric duality revisited, Inform. Process. Lett. 21 (1985) 117–122.

[15] J. Matousek, Range searching with efficient hierarchical cutting, Discrete Comput. Geom. 10 (1993) 157–182.

[16] P. Mitra, B.B. Chaudhuri, Efficiently computing the closest point to a query line, Pattern Recogn. Lett. 19 (1998) 1027–1035.

[17] A. Mukhopadhyay, Using simplicial partitions to determine a closest point to a query line, in: 14th Canadian Conference on Computational Geometry, 2002, pp. 10–12.

[18] S.C. Nandy, S. Das, P.P. Goswami, An efficient k-nearest neighbors searching algorithm for a query line, Theoret. Comput. Sci. 299 (2003) 273–288.