Theoretical Computer Science 299 (2003) 273–288

An efficient k nearest neighbors searching algorithm for a query line ☆

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

[a] *Advanced Computing and Microelectronics Unit, Indian Statistical Institute, 203 B.T. Road, Kolkata 700 108, India*

[b] *Computer Center, Calcutta University, Kolkata 700 009, India*

Abstract

We present an algorithm for finding k nearest neighbors of a given query line among a set of n points distributed arbitrarily on a two-dimensional plane. Our algorithm requires $O(n^2)$ time and $O(n^2/\log n)$ space to preprocess the given set of points, and it answers the query for a given line in $O(k+\log n)$ time, where k may also be an input at the query time. Almost a similar technique works for finding k farthest neighbors of a query line, keeping the time and space complexities invariant. We also show that if k is known at the time of preprocessing, the time and space complexities for the preprocessing can be reduced keeping the query times unchanged.

Keywords: Nearest and farthest neighbor; Query line; Arrangement; Duality

1. Introduction

Given a set $P=\{p_1, p_2, \ldots, p_n\}$ of n points arbitrarily distributed on a plane, we study the problem of finding k nearest neighbors of a query line l (in the sense of perpendicular distance). The problem of finding the nearest neighbor of a query line was initially addressed in [6]. 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 [12], with the same time and

☆ This work was done when the first author was visiting School of Information Science, Japan Advanced Institute of Science and Technology, Ishikawa, Japan.

* Corresponding author.

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

space complexities. The space complexity of the problem has recently been improved to $O(n)$ [14]. The preprocessing time of that algorithm is $O(n \log n)$, but the query time is $O(n^{0.695})$. In the same paper, it is shown that if the query line passes through a specified point, that information can be used to construct a data structure in $O(n \log n)$ time and space, so that the nearest neighbor query can be answered in $O(\log^2 n)$ time.

In this paper, we address a natural generalization of the above problem where the objective is to report k nearest neighbors of a query line in the same environment. We use geometric duality for solving this problem. Our algorithm is based on maintaining the levels of the arrangement [8] of the duals of the points in P. The preprocessing time and space required for creating the necessary data structure are $O(n^2)$ and $O(n^2/\log n)$, respectively; the query time complexity is $O(k+\log n)$, where k is an input at the query time. The same data structure can be used to report k farthest neighbors of a query line with same time complexity.

We have considered the following three constrained cases where k is known at the time of preprocessing. For all these cases the time and space complexities for preprocessing can be reduced significantly, but the query times remain same.

(i) If $k > \log n$ then for the k nearest neighbors problem, the size of the data structure can further be reduced to $O(n^2/k)$ keeping the preprocessing and query time complexities unchanged.

(ii) In particular, when the query line is known to pass through a fixed point q, we use a randomized technique to construct a data structure of size $O(kn)$ in $O(kn + \min(n \log^2 n; kn \log n))$ expected time, which answers k nearest neighbors of such a query line in $O(k+\log n)$ time. Thus, for the nearest neighbor problem (i.e., $k=1$), our algorithm is superior with respect to both space required and query time in comparison to the algorithm proposed in [14]. The preprocessing time complexity remains same as that of [14].

(iii) For the k farthest neighbors problem, the preprocessing time and space complexities can be reduced to $O(kn+n \log n)$ and $O(kn)$ respectively, and the query can be answered in $O(k + \log n)$ time.

2. Applications

Apart from being a variation of the proximity problems in computational geometry, the problem of finding k nearest/farthest neighbors is observed to be important in different applications as mentioned below.

- Consider that the points are distributed on the floor, and each point is attached with a pattern characteristics (a quantitative measure). In pattern classification and data clustering [14], the query line is considered as a partition line between two classes of patterns. Here, k nearest neighbors of the query line are considered, and the sum of squares of the pattern characteristics for *between* and *within* the partitions are analyzed to give an idea about the stability of classification.
- Another application of k nearest neighbors problem is the linear facility testing. Suppose we need to install a linear facility, e.g., pipeline, conveyor belt for plant layout, corridor for light-rail commuter system, etc., with a specified capacity k.

Here the problem is to find the sum of distances of k nearest neighbors of a query line from itself; the query line indicates the linear facility.

- The k farthest neighbors query has wide spread applications in Statistics, where the objective is to remove the farthest k elements from the query line. These are considered as outliers in the data set.

3. Geometric preliminaries

First we mention that we need to maintain an array with the points in P, sorted with respect to their x-coordinates. This requires $O(n)$ space and can be constructed in $O(n \log n)$ time. If the query line is vertical, we find the position of its x coordinate by performing a binary search in the array P. To find its k nearest neighbors, a pair of scans (towards left and right) are required in addition. So, the query time complexity is $O(k + \log n)$. It is easy to understand that k farthest neighbors of a vertical query line can be obtained in $O(k)$ time by scanning the array P from its left and right ends.

We now consider the case where the query line is non-vertical. We use geometric duality for solving these problems. Here, (i) a point $p=(a,b)$ of the primal plane is mapped to the line p': $y=ax-b$ in the dual plane, and (ii) a non-vertical line l: $y=mx-c$ of the primal plane is mapped to the point $l'=(m,c)$ in the dual plane. The incidence and order relationships between a point p and a line l in the primal plane remain preserved among their duals in the dual plane [12].

Let H be the set of dual lines corresponding to the points in P. Let $\mathscr{A}(H)$ denote the arrangement of the set of lines H. The number of vertices, edges and faces in $\mathscr{A}(H)$ are all $O(n^2)$ [8]. Given a query line l, the problem of finding its nearest and farthest point can be solved as follows:

Nearest-neighbor algorithm: Use the point location algorithm of [9] to locate the cell of $\mathscr{A}(H)$ containing l' (the dual of the line l) in $O(\log n)$ time. As the cells of the arrangement $\mathscr{A}(H)$ are split into trapezoids, the lines p_i' and p_j' lying vertically above and below l', can be found in constant time. The distance of a point p and the line l in the primal plane can be obtained from the dual plane as follows:

- Draw a vertical line from the point l' which meets the line p' at a point $\alpha(l', p')$ in the dual plane. The *perpendicular distance* of the point p and the line l in the primal plane is equal to $d(l, \alpha(l', p'))/\sqrt{1+(x(l'))^2}$, where $d(\cdot,\cdot)$ denotes the distance between two points, and $x(l')$ is the x-coordinate of the point l'.

Thus, the point nearest to l in the primal plane will be any one of p_i or p_j depending on whether $d(l', \alpha(l', p_i')) <$ or $> d(l', \alpha(l', p_j'))$. The preprocessing time and space required for creating and storing $\mathscr{A}(H)$ are both $O(n^2)$ [9]. From now onwards, $d(l', \alpha(l', p'))$ will be referred as the *vertical distance* of the line p' from the point l'.

Farthest-neighbor algorithm: We construct the lower and upper envelopes of the lines in H, and store their vertices in two different arrays, say A_1 and A_2. This requires $O(n \log n)$ time [11]. Now, given the line l, or equivalently the dual point l', we draw a vertical line at $x(l')$ which hits the edges $e_a \in A_1$ and $e_b \in A_2$. Edges e_a and e_b can be located in $O(\log n)$ time. If e_a and e_b are, respectively, portions of p_i' and p_j', then the farthest neighbor of l is either p_i or p_j which can be identified easily. Thus for

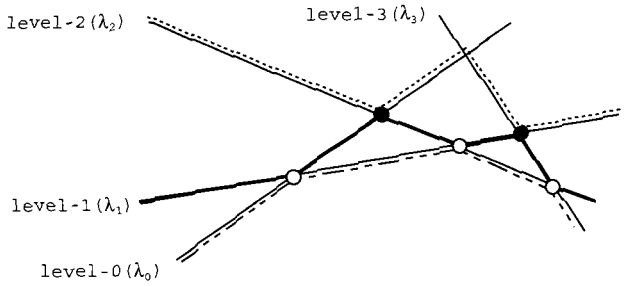

Fig. 1. Demonstration of levels in an arrangement of lines.

the farthest neighbor problem, the preprocessing time and space required are $O(n \log n)$ and $O(n)$, respectively, and the query can be answered in $O(\log n)$ time.

We follow almost the same approach for locating k nearest/farthest neighbors of a query line. The data structure used in our algorithm stores different levels of the arrangement $\mathscr{A}(H)$, as stated below.

Definition 1 (Edelsbrunner [8]). A point π in the dual plane is at level θ $(0 \leqslant \theta \leqslant n)$ if there are exactly θ lines in H that lie strictly below π. The θ-level of $\mathscr{A}(H)$ is the closure of a set of points on the lines of H whose levels are exactly θ in $\mathscr{A}(H)$, 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 $\mathscr{A}(H)$ appears in two consecutive levels, and each edge of $\mathscr{A}(H)$ appears in exactly one level. In Fig. 1, a demonstration of levels in an arrangement $\mathscr{A}(H)$ is shown. The thick chain represents λ_1. Among the vertices of λ_1, those marked with empty circles appear in λ_0, and those marked with black circles appear in λ_2.

4. Algorithm

In order to make the presentation clear to the reader, we first describe a preprocessing strategy which can be performed in $O(n^2)$ time and using $O(n^2)$ space, and it reports k nearest/farthest neighbors of a query line in $O(k + \log n)$ time. Next, we modify our data structure to achieve $O(n^2/\log n)$ space bound; the preprocessing and query algorithms are also modified accordingly but the time complexities remain unchanged.

4.1. Data structure

We create a balanced binary search tree T, called the *primary structure*, whose nodes correspond to the levels $\{\theta \mid \theta = 1, \ldots, n\}$ of the arrangement $\mathscr{A}(H)$. Each node, representing a level θ, is attached with a secondary structure, which is a linear array

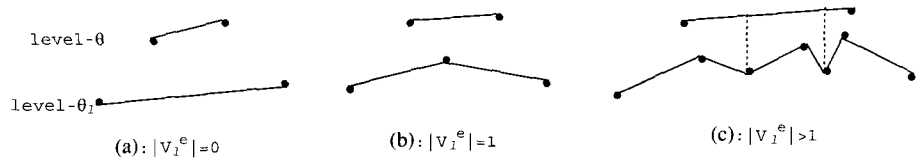

Fig. 2. Augmentation of *secondary* structure.

containing the vertices and edges of λ_θ in a left to right order. From now onwards, the secondary structure of a node containing level θ will be referred as λ_θ.

To facilitate our search process, we follow a technique similar to *fractional cascading* [5] for augmenting the secondary structures of all the non-leaf nodes in T as described below.

Consider an edge $e \in \lambda_\theta$. Let θ_l be the level attached to the left child of the level θ in T. Let V_l^e be the set of vertices of λ_{θ_l} whose projections on the x-axis are completely covered by the projection of e on the x-axis. If $|V_l^e| > 1$ then we leave the leftmost vertex of V_l^e and vertically project the next vertex on e, and do the same for every alternate vertices of V_l^e as shown in Fig. 2. All these projections create new vertices at level θ. We consider all the edges of λ_θ, and for each edge $e \in \lambda_\theta$, we select a subset of vertices of λ_{θ_l} whose projections on e create new vertices at level θ. Similarly, a selected subset of vertices of λ_{θ_r} (the right child of θ) will also contribute new vertices at level θ. Now we have the following lemma.

Lemma 1. *After the augmentation step*

(i) *the projection of an edge of λ_θ on the x-axis can overlap on the projections of at most two edges of λ_{θ_l} and at most two edges of λ_{θ_r} on the x-axis.*

(ii) *If the projection of an edge $e \in \lambda_\theta$ on x-axis overlaps on that of two edges of λ_{θ_l}, they share a common vertex of λ_{θ_l}. The same result holds for λ_{θ_r} also.*

(iii) *The number of edges at level θ is increased to at most $|E_\theta| + |E_{\theta_l}| + |E_{\theta_r}|/2$, where E_θ is the set of edges in λ_θ.*

Proof. (i) Consider the projections of a selected subset of vertices of λ_{θ_l} on the edges at level θ. For each edge $e \in \lambda_\theta$, the following three situations may arise.

$|V_l^e| = 0$, i.e., the edge e is completely covered by an edge at level θ_l (Fig. 2(a)).

$|V_l^e| = 1$, i.e., the edge e overlaps on exactly two edges at level θ_l (Fig. 2(b)).

$|V_l^e| > 1$. In this case the truth of the lemma follows as we have projected every alternate vertex of V_l^e on e (Fig. 2(c)), and edge e is split into parts after the augmentation.

The same result follows for the projections of the vertices of λ_{θ_r} on the edges of λ_θ.

(ii) Follows from part (i) of this lemma.

(iii) Follows from the fact that at most half of the vertices of each of λ_{θ_l} and λ_{θ_r} are projected on the polychain at level θ. □

From the original secondary structures of all nodes in T, we create new secondary structures as follows:

- The secondary structures for all the nodes appearing at the leaf level of T will remain unchanged. We propagate a selected subset of vertices appearing in the secondary structure of each leaf node to that of its parent as stated above, and mark the leaf nodes as processed.
- We select a non-leaf node which is not yet processed but both of its successors have already been processed, and construct a linear array with the enhanced set of vertices and edges formed by the original vertices that are present in its existing secondary structure, and the vertices contributed by its both left and right children due to their projections. The members in the array are ordered from left to right. This new array will now serve the role of the secondary structure. We mark the node as processed. If the node is not the root, we propagate a selected subset of its vertices (after augmentation) to its parent.
- This method of propagation of vertices is continued in a bottom-up manner until the root of T is processed.

Lemma 1 remains valid for all the edges appearing in the new secondary structure of all non-leaf nodes. The augmented structure will be referred as $\mathscr{A}^*(H)$. Its primary structure T remains the same; from now onwards, λ_θ will denote the modified secondary structure of node θ.

Lemma 2. *The total number of vertices in the secondary structures of all the levels of* $\mathscr{A}^*(H)$ *is* $O(n^2)$.

Proof. All the vertices of $\mathscr{A}(H)$ contribute to $\mathscr{A}^*(H)$. We now show that the number of vertices created in the secondary structures of all the nodes in T due to the augmentation remains $O(n^2)$. Here we use the term *layer* to denote the different depth levels of the tree T. For the sake of notational simplicity, let $h=(\lceil \log n \rceil)$. The tree T has h layers, layer-1 corresponds to the root of T and the layer indices increase as we proceed towards the leaves of T.

Let n_i and n_i^* denote the number of vertices present in the secondary structures of all the nodes in the ith layer of the tree T before and after the augmentation process, respectively. Since the secondary structures of the leaf nodes of T are not augmented, $n_h^* = n_h$. At most $n_h/2$ nodes are propagated to the secondary structures of the nodes appearing in the $(h-1)$th layer of T. So, $n_{h-1}^* = n_{h-1} + n_h/2 = n_{h-1} + n_h^*/2$. Proceeding in a similar manner, $n_i^* = n_i + n_{i+1}^*/2$. Thus, the total number of vertices in $\mathscr{A}^*(H)$ is $\sum_{i=1}^{h} n_i^* = n_h + (n_{h-1} + n_h/2) + (n_{h-2} + n_{h-1}/2 + n_h/4) + \cdots + (n_1 + n_2/2 + n_3/4 + \cdots + n_h/2^{h-1}) \leqslant 2\sum_{i=1}^{h} n_i = O(n^2)$. □

Each edge in the secondary structure of a non-leaf node is attached with two pointers, namely Π_1 and Π_2. Let θ be a non-leaf node, and θ_l and θ_r be its left and right children, respectively. By Lemma 1, the projection of an edge $e \in \lambda_\theta$ overlaps on that of at most two edges of both λ_{θ_l} and λ_{θ_r}. If e overlaps on one edge, say $e^* \in \lambda_{\theta_l}$ then the Π_1 pointer of edge e, points to e^* . If e overlaps on two edges, say $e^*, e^{**} \in \lambda_{\theta_l}$ then Π_1 pointer of edge e points to the vertex common to e^* and e^{**}. The pointer Π_2 of edge

e is set to point an edge or a vertex common to two adjacent edges of λ_{θ_r} in a similar manner.

4.2. Sketch of the query algorithms

Given a query line l, we compute its dual point l'. Next, we identify a pair of adjacent levels of $\mathscr{A}^*(H)$ and their corresponding edges which appear vertically above and below l', using the procedure described below.

Definition 2. Let e be an edge in $\mathscr{A}^*(H)$ and α be an arbitrary point in the dual plane. The coordinates of the left and right end points of e be (x_e^l, y_e^l) and (x_e^r, y_e^r), respectively, and the coordinates of the point α be (x_α, y_α). The edge e is said to *span horizontally on the point* α if the projection of the edge e on the x-axis contains the projection of the point α on x-axis, i.e., $x_e^l \leqslant x_\alpha \leqslant x_e^r$.

We start from the root of T. Let θ_{root} be the level (of $\mathscr{A}^*(H)$) attached to it. We use binary search to locate the edge $e \in \lambda_{\theta_{\text{root}}}$ which spans horizontally on l'. If l' is below e we proceed towards the left child (θ_l) of θ_{root} in T; otherwise, we proceed towards the right child (θ_r). If search proceeds towards θ_l, we can find the edge $e^* \in \lambda_{\theta_l}$ spanning horizontally on l' in O(1) time using the pointer Π_1 attached to the edge e. Similarly, we use the pointer Π_2 for the same purpose if search proceeds towards θ_r. Proceeding in a similarly manner, we can identify a level θ in the leaf layer of T, and an edge in its secondary structure λ_θ which is just above or below l'. The other edge defining the cell can easily be identified from $\lambda_{\theta-1}$ or $\lambda_{\theta+1}$. This step requires $O(\log n)$ time. We shall refer this traversal in T as *forward traversal.*

4.2.1. Reporting of k nearest neighbors

Let $e_a \in \lambda_\theta$ and $e_b \in \lambda_{\theta-1}$ be the two edges which are vertically above and below l', respectively, in the cell (of $\mathscr{A}^*(H)$) containing l'. We compute the vertical distances of e_a and e_b from l', and report the nearest one. If e_a is closer to l' than e_b, we need to reach an edge of $\lambda_{\theta+1}$ whose horizontal span contains l'. On the other hand, if e_b is closer to l' than e_a, we need to reach such an edge of $\lambda_{\theta-2}$.

In order to reach an appropriate edge in the inorder predecessor or successor of a node during the reporting, we maintain two stacks S_1 and S_2. They contain the edges (along with the level-id) of $\mathscr{A}^*(H)$ through which the search proceeded from the root to level $\theta-1$ and θ, respectively. Initially, these two stacks are prepared during forward traversal. At the time of reporting, these stacks will dynamically change as described in the proof of following lemma.

Lemma 3. *After locating a pair of edges in two consecutive levels of* $\mathscr{A}^*(H)$, *say* $\theta-1$ *and* θ, *whose horizontal spans contain* l', *the edges in* k *levels vertically below* $\theta-1$ (*vertically above* θ) *can be reported in* $O(k+\log n)$ *time.*

Proof. Without loss of generality, we consider the method of visiting of the edges in k consecutive levels above the level θ whose horizontal span contains l'.

If the level $\theta+1$, which is the inorder successor of level θ in T, is

(i) in its right subtree, then we traverse all the levels that appear along the path from θ to $\theta+1$ in T. In each move, we use pointers Π_1 and Π_2 to reach an edge in the next layer whose horizontal span contains l' in constant time. We need to store all these edges in the S_2 stack for backtracking, if necessary.

(ii) in some predecessor layer, then we may need to backtrack along the path through which we reached from $\theta+1$ to θ during forward traversal. This can be done by popping elements from S_2 stack until we get an edge of level $\theta+1$. Let the number of elements popped be δ.

Note that, after visiting level $\theta+1$, if it needs to proceed to the level $\theta+2$, we again have to move forward δ layers towards leaf. During this forward traversal, the edges on that path will be pushed in the S_2 stack. But such a forward movement may again be required after visiting all the levels in the right subtree rooted at level $\theta+1$. Thus apart from reporting, this extra traversal in T may be required at most twice, and the length of the path may be at most $O(\log n)$. This implies that, each of the k nearest lines of l' can be reported in amortized $O(1)$ time.

In order to visit the edges in k consecutive levels below $\theta-1$, whose horizontal span contains l', we need to use the stack S_1 in the same manner. □

Thus Lemmas 2 and 3 lead to the following result stating the time and space complexities of our algorithm.

Theorem 1. *Given a set of points on a plane, they can be preprocessed in* $O(n^2)$ *time and space such that the problem of reporting k nearest neighbors of a given query line can be solved* $O(k+\log n)$ *time.*

4.2.2. Reporting of k farthest neighbors

In the farthest neighbor problem, we find two edges $e_a \in \lambda_1$ and $e_b \in \lambda_n$, whose horizontal span contains l', by traversing the tree T from its root to the leaves containing levels 1 and n. The farthest neighbor of l' is either the line containing e_a or the line containing e_b as mentioned in Section 3. In order to get the next farthest neighbor, the search progresses in T to the inorder successor of level 1 or the inorder predecessor of level n depending on which one is reported currently. The process continues until k lines are reported. Here also we need to maintain two stacks S_1 and S_2, whose role is same as that of the earlier problem. In each level, the desired edge can be located in amortized constant time using these stacks.

Theorem 2. *Given a set of points on a plane, they can be preprocessed in* $O(n^2)$ *time and space such that the problem of reporting k farthest neighbors for a given query line can be solved in* $O(k+\log n)$ *time.*

In this connection it needs to mention that, an alternative method for finding the k-nearest neighbors of a query line can be devised using the $(1/n)$-*cutting tree* in the dual plane [2,13] keeping the preprocessing time and space complexities $O(n^2)$ and the

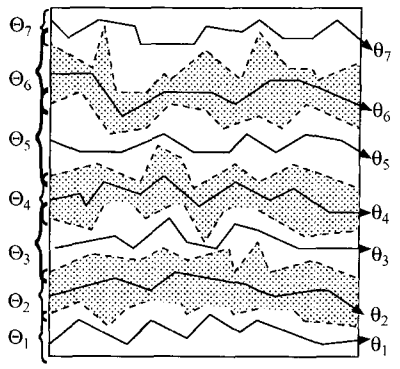

Fig. 3. Demonstration of Lemma 4.

query time complexity $O(k + \log n)$. But this method cannot solve the other problems addressed in this paper.

We now discuss some modification of our existing data structure to achieve better space complexity.

4.3. Further refinement

The following result helps in reducing the space complexity of the preprocessed data structure keeping the preprocessing and query time complexities invariant.

Lemma 4 (Pach and Agarwal [16, Lemma 11.4]). *Let $\Theta_1, \Theta_2, \ldots, \Theta_u$ be disjoint collections of levels in an arrangement of n lines. If each Θ_i contains at least v levels, then we can pick a level $\theta_i \in \Theta_i$, $1 \leqslant i \leqslant u$, such that $\sum_{i=1}^{u} |E_{\theta_i}| \leqslant n^2/v$. Here E_θ denotes the set of edges at level θ.*

We now describe a method of reporting m nearest/farthest neighbors of a query line. In order to answer the queries in different cases (as mentioned in Section 1), we have to choose the value of m appropriately.

Consider $\lceil n/m \rceil$ disjoint sets of levels, namely $\Theta_1, \Theta_2, \ldots, \Theta_{\lceil n/m \rceil}$, Θ_i is a collection of m consecutive levels $\{(i-1)m+1, (i-1)m+2, \ldots, im\}$ for $i = 1, \ldots, \lfloor n/m \rfloor$, and $\Theta_{\lceil n/m \rceil}$ consists of a set of $(n - m * \lfloor n/m \rfloor)$ consecutive levels $\{m\lfloor n/m \rfloor + 1, m\lfloor n/m \rfloor + 2, \ldots, n\}$. The partitioning of levels is demonstrated in Fig. 3. We choose levels $\{\theta_i,\ i = 1, \ldots, \lceil n/m \rceil\}$, where $\theta_i \in \Theta_i$ and $|\lambda_{\theta_i}| = \min_{j \in \Theta_i} |\lambda_j|$, and construct the data structure as described below.

4.3.1. Data structure

Our primary structure is a height balanced tree T whose nodes correspond to the levels $\{\theta_i,\ i = 1, \ldots, \lceil n/m \rceil\}$. The secondary structure attached with level θ_i is an array containing the vertices and edges on the polychain λ_{θ_i}. The augmentation step remains same as described in Section 4.1. The role of the two pointers Π_1 and Π_2, attached

with each edge of the augmented structure, remain same as mentioned earlier. The modified data structure will be referred as $\mathscr{B}^*$.

Lemma 5. *The total number of vertices generated in all the levels* $\lambda_{\theta_1}, \lambda_{\theta_2}, \ldots, \lambda_{\theta_{\lceil n/m \rceil}}$ *after the augmentation step is* $O(n^2/m)$ *in the worst case.*

Proof. See Lemma 4 and the proof of Lemma 2. □

Consider a vertex $v \in \lambda_{\theta_i}$, and draw a vertical line ρ from vertex v upwards till it hits the polychain $\lambda_{\theta_{i+1}}$. Let $Q(v)$ be the set of lines of H that are intersected by ρ. As we are not storing all the levels, we need to maintain $Q(v)$ with each vertex $v \in \mathscr{B}^*$. This will facilitate the query answering. Note that, $0 \leqslant |Q(v)| \leqslant 2m$ ($|Q(v)| = 0$ is attained if θ_i is the highest level in Θ_i, and θ_{i+1} is the lowest level in Θ_{i+1}; similarly, $|Q(v)| = 2m$ is attained if θ_i is the lowest level in Θ_i and θ_{i+1} is the highest level in Θ_{i+1}). Thus, the total space required for storing $Q(\cdot)$ for all the vertices of $\mathscr{B}^*$ may be $O(n^2)$ in the worst case (see Lemma 5). In order to reduce the space complexity, we store $Q(\cdot)$ with a selected subset of vertices in each level of $\mathscr{B}^*$. Let $V_{\theta_i} = \{v_0, v_1, \ldots\}$ be the set of vertices of λ_{θ_i} after the augmentation step. We split V_{θ_i} into two subsets $V^1_{\theta_i} = \{v_0, v_m, v_{2m}, v_{3m}, \ldots\}$ and $V^2_{\theta_i} = V_{\theta_i} - V^1_{\theta_i}$. The number of vertices in $V^1_{\theta_i}$ is $O(\lfloor n/m \rfloor)$. Each of the vertices in the subset $V^1_{\theta_i}$ is attached with the list $Q(\cdot)$. While answering a query, if $Q(\cdot)$ is required for a vertex $v_j \in V^2_{\theta_i}$, it will be computed online from that of its nearest vertex in $V^1_{\theta_i}$ (i.e., the vertex $v_{m\lfloor j/m \rfloor}$). For this purpose, we attach two scalar information with each vertex $v_j \in V^2_{\theta_i}$: (i) a pointer Φ which points to the vertex $v_{m\lfloor j/m \rfloor} (\in V^1_{\theta_i})$, and (ii) an integer Ψ containing either $\theta_i - 1$ or $\theta_i + 1$ in which v_j appears in addition to level θ_i in $\mathscr{A}(H)$. If the vertex is created due to the augmentation, it cannot appear in two levels of $\mathscr{A}(H)$; so the Ψ field attached to that vertex contains 0.

Each edge $e \in \mathscr{B}^*$ is attached with the identifier of the line (*line-id*) contributing that edge. If e is a part of the line p'_i (dual of the point p_i), its *line-id* is i.

In addition to the data structure $\mathscr{B}^*$, we need to maintain an array L of size n which contains all the lines in H. Its elements are ordered arbitrarily. Each element contains (i) a pointer field and (ii) a mark bit. The pointer field is used to point the corresponding element in a temporary list *TEMP* which is used during the preprocessing, and will be defined in the next subsection. The mark bit is used for query answering.

Lemma 6. *The space required for storing the additional information attached with all the vertices and edges in* $\mathscr{B}^*$ *is* $O(n^2/m)$ *in the worst case.*

Proof. $|V^1_{\theta_i}| = \lceil |V_{\theta_i}|/m \rceil \leqslant |V_{\theta_i}|/m + 1$. The total number of vertices with which $Q(\cdot)$ is attached, is $\sum_{i=1}^{\lceil n/m \rceil} (|V_{\theta_i}|/m + 1) = n^2/m^2 + n/m$ in the worst case (by Lemma 5). The lemma follows from the fact that the size of $Q(\cdot)$ for each of these vertices is less than $2m$. It needs to mention that the total extra space consumed for the set of vertices in $\{V^2_{\theta_i},\ i = 1, \ldots, \lceil n/m \rceil\}$ and for all edges in $\mathscr{B}^*$ is $O(n^2/m)$ in the worst case. □

4.3.2. Preprocessing

We construct the aforesaid data structure in three major steps:

Step A1: Using topological line sweep [1], we can construct the levels of the arrangement in $O(n^2)$ time. By observing the number of vertices in each level, we can identify $\{\theta_1, \theta_2, \ldots, \theta_{\lceil n/m \rceil}\}$. Note that, we don't need to store the levels explicitly in this step, so $O(n)$ space is enough.

Step A2: Again, we perform topological line sweep [1] to explicitly identify the vertices and edges on the polychains $\{\lambda_0, \lambda_{\theta_1}, \lambda_{\theta_2}, \ldots, \lambda_{\theta_{\lceil n/m \rceil}}, \lambda_n\}$; here λ_0 and λ_n represent the lower and upper envelopes of H, respectively. With each edge, its *line-id* is attached. We then augment the data structure as described in Section 4.1 to get the desired structure $\mathscr{B}^*$. Next, we consider each level θ_i, and identify two sets of vertices $V^1_{\theta_i}$ and $V^2_{\theta_i}$. The pointers attached to the vertices in $V^2_{\theta_i}$ are appropriately set. The entire step can be completed in $O(n^2)$ time.

Step A3: In the last step, we process each pair of consecutive levels (θ_i, θ_{i+1}) for $i = 0, 1, \ldots, \lceil n/m \rceil$, and prepare list $Q(v)$ for each vertex $v \in V^1_{\theta_i}$. The processing of this step is done using the following substeps.

Step A3.1: [Initialization step]

The pointers attached with all the members in the array L are initialized to NULL.

Create a new list by merging the vertices of the polychains λ_{θ_i} and $\lambda_{\theta_{i+1}}$ in increasing order of their x-coordinates.

Consider a vertical line segment ρ at the leftmost vertex of the new list between the polychains λ_{θ_i} and $\lambda_{\theta_{i+1}}$, and create a temporary list *TEMP* with the lines in array L that intersect ρ. The pointer field attached with each element in the array L is set with the address of the corresponding element in the list *TEMP*.

Step A3.2: We process the vertices of the list created in Step A3.1 by sweeping the vertical line segment ρ towards right. During the sweep, the two end points of ρ will always touch λ_{θ_i} and $\lambda_{\theta_{i+1}}$, and *TEMP* contains the lines of L that intersects ρ. For a vertex $v \in \mathscr{A}(H)$, let l_{below} and l_{above} be the *line-ids* of two edges which are incident to v from left as shown in Fig. 4. Now, the following four situations need to be considered.

(a) If v appears in levels θ_i and $\theta_i - 1$ (see Fig. 4(a)), no change in the list *TEMP* is required.

(b) If v appears in levels θ_i and $\theta_i + 1$ (see Fig. 4(b)), then after processing v_i, the line l_{above} leaves ρ, and l_{below} appears on ρ. We use *line-ids* attached to l_{below} and l_{above} to reach the corresponding elements of array L. The pointers attached to these two lines help in accessing them in the list *TEMP*. *TEMP* is updated by deleting l_{above} and adding l_{below}. Finally, the pointers attached to l_{below} and l_{above} in the array L are updated accordingly. This step requires $O(1)$ time.

(c) If v appears in levels θ_{i+1} and $\theta_{i+1} - 1$ (see Fig. 4(c)), l_{below} goes out and l_{above} enters in the list *TEMP*. Here also, the necessary updates can be done in $O(1)$ time.

(d) If v appears in levels θ_{i+1} and $\theta_{i+1} + 1$ (see Fig. 4(d)) no change in *TEMP* is required.

Step A3.3: If $v \in V^1_{\theta_i}$, a copy of the list TEMP is attached with the vertex v as $Q(v)$

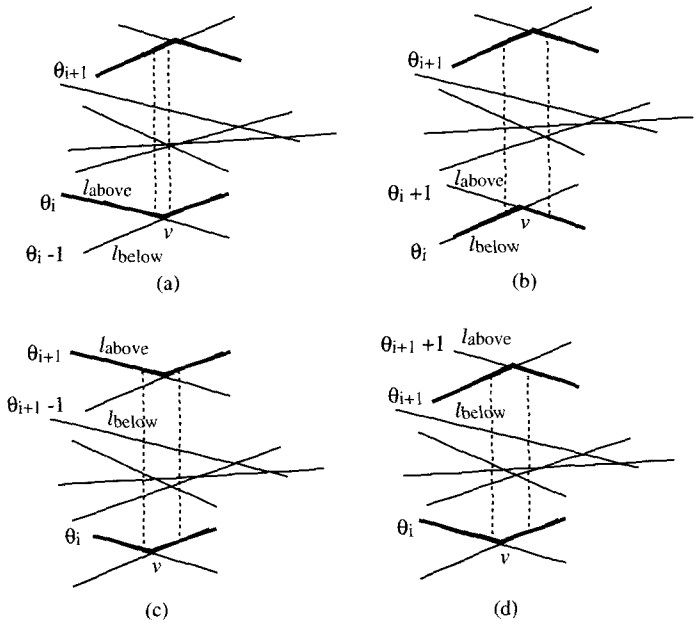

Fig. 4. Updating of *TEMP* list.

Lemma 7. *The preprocessing step requires* $O(n^2)$ *time in the worst case.*

Proof. Steps Al and A2 require $O(n^2)$ time. The total time complexity of Step A3 for all pairs of levels $\{(\theta_i, \theta_{i+1}), i = 1, \ldots, \lceil n/m \rceil\}$ is calculated as follows:

(i) the initialization process (Step A3.1) requires $O(n^2)$ time in total;
(ii) $O(n^2/m)$ vertices are considered in total (Lemma 6) in Step A3.2, and processing each vertex needs $O(1)$ time, and
(iii) Step A3.3 needs to be executed for $O(n^2/m^2)$ vertices in total. For each of the vertices, we need to create $Q(\cdot)$ by copying the list TEMP in it. As the number of lines in TEMP at any instant of time can never exceed $2m$, Step A3.3 takes $O(n^2/m)$ time during the whole execution. □

4.3.3. Query

Given a query line l, we compute its dual point l', and locate the edges of $\mathscr{B}^*$ which are just above and below l' using the same technique as described in Section 4.2. Let l' lies in the region bounded by the polychains $\lambda_{\theta_{i+1}}$ and λ_{θ_i}. We draw a vertical line segment ρ^* from l' upwards till it hits the polychain at level $\min(\theta_{i+3}, n)$, and another line ρ^{**} from l' downwards till it hits the polychain at level $\max(1, \theta_{i-2})$. Here also, we need to create an array A which contains the lines of H that are intersected by ρ^* and ρ^{**}. The m nearest lines of l' are obtained by searching the members of A with respect to their vertical distances from l'.

Lemma 8. $m \leqslant |A| \leqslant 8m$.

Proof. Consider the two extreme situations: (i) θ_{i+1} is the maximum level in Θ_{i+1} and θ_{i+3} is the minimum level in Θ_{i+3} and (ii) θ_{i+1} is the minimum level in Θ_{i+1} and θ_{i+3} is the maximum level in Θ_{i+3}. The number of lines hitting ρ^* between λ_{θ_i+1} and λ_{θ_i+3} in case (i) is at least m and that in case (ii) is at most $3m$. Same result holds for the set of lines intersected by ρ^{**} between the levels θ_i and θ_{i-2}. The total number of lines hitting ρ^* and ρ^{**} between λ_{θ_i} and $\lambda_{\theta_{i+1}}$ may be atmost $2m$. Hence the result follows. □

We consider six pseudo vertices $\{w_j,\ j=1,2,\ldots,i+3\}$, where w_j is the point of intersection of the polychains λ_{θ_j} with the vertical line drawn at l', i.e., with either of ρ^* and ρ^{**}. Lemma 8 indicates that, in order to find k nearest neighbors of l', we need to consider $Q(w_{i-2}), Q(w_{i-1}), Q(w_i), Q(w_{i+1})$ and $Q(w_{i+2})$.

Lemma 9. *$Q(w_j)$ can be constructed in* O(m) *time.*

Proof. Let $v\in\lambda_{\theta_j}$ is closest and to the left of w_j. If $v\in V^1_{\theta_j}$ then $Q(w_j)=Q(v)$. If $v\in V^2_{\theta_j}$ then we choose a vertex $v^*\in V^1_{\theta_j}$ which is closest and to the left of v. v^* can be reached from v using the pointer Φ attached to v. We initialize $Q(w_j)$ by $Q(v^*)$ and process the vertices of λ_{θ_j} and λ_{θ_j+1} starting from vertex v^* towards right until v is reached. While processing each vertex we use Step A3.2 (see Section 4.3.2) to update $Q(w_j)$. The result follows from the fact that the number of vertices to be processed between v^* and v is at most m, and processing each vertex requires O(1) time. □

The major steps in the query algorithm are listed below.

Step Bl: Locate the edges of $\mathscr{B}^*$ which are just above and below the point l' (dual of the query line l) as described in Section 4.2.

Step B2: Use the method described in Section 4.2.1 (using Π_1 and Π_2 pointers attached with the edges of $\mathscr{B}^*$) to obtain the points $w_j\in\lambda_{\theta_j}$ for $j=i-2,i-1,i,i+1,i+2,i+3$.

Step B3: Create $Q(w_{i-2}), Q(w_{i-1}), Q(w_i), Q(w_{i+1})$, and $Q(w_{i+2})$.

Step B4: Create the array A with the members in $Q(w_{i-2})\cup Q(w_{i-1})\cup Q(w_i)\cup Q(w_{i+1})\cup Q(w_{i+2})$. The mark bit of a line is checked prior to its inclusion in A, and after the inclusion of a line in array A its mark bit is set. This ensures that no line is included more than once in A.

Step B5: Compute the vertical distances of all the lines in A from the point l', and find exactly m lines closest to l' using *median find algorithm* [7] in O(m) time.

Step B6: Next, each element in array A is considered; its *line-id* is used to access the same line in the array L in O(1) time. Finally, the mark bit attached to it in the array L is reset to zero. This step is required for the subsequent query on the same data structure.

Lemma 10. *For a given set of n points we can construct a data structure of size* O(n^2/m) *in* O(n^2) *time, which can report m-nearest neighbors of a query line l in* O($m+\log n$) *time in the worst case.*

Proof. The preprocessing time and space complexities follow from Lemmata 6 and 7, respectively. In the query algorithm, Step B1 requires $O(\log n)$ time. As mentioned in the algorithm, all other steps can be completed in $O(m)$ time. □

Theorem 3. *If k is not known prior to the preprocessing then for a given set of n points we can construct a data structure of size $O(n^2/\log n)$ in $O(n^2)$ time, and it can report k nearest/farthest neighbors of an arbitrary query line in $O(k+\log n)$ time.*

Proof. We choose $m=\log n$ to achieve the space complexity result. In order to report k nearest neighbors, we need to consider the following two situations:

- If $k<\log n$, then we need to apply the query algorithm only once; so the query time complexity follows from Lemma 10.
- If $k>\log n$, then we need to locate l' (the dual of line l) in the appropriate cell of $\mathscr{B}^*$ only once. In order to report k nearest neighbors, we may need to apply Step B2–B5 of the query algorithm at most $k/\log n$ times. Each application returns at least $\log n$ points in $O(\log n)$ time. Hence the query time complexity follows. □

5. Constrained query: k is known prior to the preprocessing

5.1. Reporting of k nearest neighbors of a query line

The following theorem states that if $k>\log n$, then the space complexity of the k nearest neighbors problem can be improved further.

Theorem 4. *If k is known prior to the preprocessing and $k>\log n$, then for the given set of n points we can construct a data structure of size $O(n^2/k)$ in $O(n^2)$ time, and it can report k nearest neighbors of an arbitrary query line in $O(k+\log n)$ time.*

Proof. We choose $m=k$ to achieve the preprocessing time and space complexity results. Only a single application of the aforesaid query algorithm returns k nearest neighbors in $O(k+\log n)$ time. □

5.2. Reporting of k nearest neighbors when query line passes through a fixed point

We show that if the query line l passes through a specified point q, then the preprocessing time and space complexities may further be reduced. Here, the point l' (dual of the line l) will always lie on the line $h=q'$ (dual of the point q).

We split each line $h_i\in H$ into two parts h_i^a and h_i^b, where h_i^a is the portion of h_i above h, and h_i^b is the portion of h_i below h. Let $H^a=\{h_1^a,h_2^a,\ldots,h_n^a\}$ and $H^b=\{h_1^b,h_2^b,\ldots,h_n^b\}$. We use $\mathscr{A}_{\leqslant k}(H^a)$ and $\mathscr{A}_{\leqslant k}(H^b)$ to denote $(\leqslant k)$-levels above and below h, respectively.

In [18], the *zone theorem* for line arrangement [15] is used to show that the complexity of both $\mathscr{A}_{\leqslant k}(H^a)$ and $\mathscr{A}_{\leqslant k}(H^b)$ are $O(nk)$. A randomized algorithm is proposed in [18] which computes $\mathscr{A}_{\leqslant k}(H^a)$ and $\mathscr{A}_{\leqslant k}(H^b)$ in $O(kn+\min(n\log^2 n, kn\log n))$ expected time.

Next, we compute the augmented data structure $\mathscr{A}^*_{\leqslant k}(H^a)$ and $\mathscr{A}^*_{\leqslant k}(H^b)$ as follows: start from level k and proceed up to level 1; at each level, select the set of alternate vertices and project them into its next lower level, and create new vertices at that level. The polychain (secondary structure) at level θ of $\mathscr{A}^*_{\leqslant k}(H^a)$ is denoted as λ^a_θ and that of $\mathscr{A}^*_{\leqslant k}(H^b)$ is denoted as λ^b_θ. We attach a pointer Π with each edge e, which points to the vertex/edge in its next higher level that is contained in the horizontal span of edge e. The size of the augmented structures will remain $O(nk)$ (see the proof of Lemma 2).

Given a query line l, its dual point l' lies on the line h. Its nearest neighbor is one of the edges $e_1 \in \lambda^a_1$ and $e_2 \in \lambda^b_1$, and they are obtained using binary search in the respective arrays. To report the next nearest neighbor, we move to the next level of either $\mathscr{A}^*_{\leqslant k}(H^a)$ or $\mathscr{A}^*_{\leqslant k}(H^b)$ using the Π pointer attached to e_1 or e_2. This process is repeated until k lines are reported.

Theorem 5. *If the query line is known to pass through a specified point q, and k is known in advance, then for a given set of n points we can construct a data structure of size $O(nk)$ in $O(nk + \min(n\log^2 n, kn\log n))$ expected time, which can report k nearest neighbors of such a query line in $O(k + \log n)$ time.*

5.3. Reporting of k farthest neighbors of a query line

In this case, we need to maintain only (i) k levels from bottom starting from level-1 up to the level-k, denoted by $\mathscr{A}_{\leqslant k}$, and (ii) k levels at the top starting from level-$(n-k+1)$ up to level-n, denoted by $\mathscr{A}_{\geqslant(n-k+1)}$, of the arrangement $\mathscr{A}(H)$. The number of edges in the $\mathscr{A}_{\leqslant k}$ of an arrangement H of n lines in the plane is $O(nk)$ (see Corollary 5.17 of [17]) and it can be computed in $O(nk + n\log n)$ time [10].

After constructing $\mathscr{A}_{\leqslant k}$ we start augmenting the data structure from level-k and proceed up to level-1. The process of augmentation, and the role of the pointer attached to each edge of the augmented data structure are same as that of the earlier problem. A similar method is followed to augment $\mathscr{A}_{\geqslant(n-k+1)}$; it starts from level-$(n-k+1)$ and proceed up to level-n.

After augmentation, the size of the data structures will remain $O(nk)$. As mentioned in Section 3, the farthest neighbor of a point l' (dual of the query line l) is either an edge at level-1 or an edge at level-n, and these can be located using binary search. The remaining $k-1$ farthest neighbors are obtained in a similar manner as described for the previous problem. Thus we have the following theorem:

Theorem 6. *If k is known in advance, the given set of n points can be preprocessed in $O(nk + n\log n)$ time and $O(nk)$ space, such that for any arbitrary query line, its k farthest neighbors can be reported in $O(k + \log n)$ time.*

6. Conclusion

The problem of finding k nearest neighbors of a query line among a set of points distributed arbitrarily on a two-dimensional plane is studied. Our preprocessing scheme

creates a data structure of size $O(n^2/\log n)$ in $O(n^2)$ time, and the query can be answered in $O(k+\log n)$ time, where k may be specified at query time. Some restricted cases of the problem are also studied when k is known prior to the preprocessing.

The average space complexity of the problem may be improved to $O((n/k)^2)$ using (i) the idea of ε-approximation of the labeling of arrangement [3] or (ii) the result in Theorem 11.6 [17]. But the preprocessing time may be worse than that of ours (see [4, Theorem 14]).

Acknowledgements

The authors wish to acknowledge the Prof. T. Asano and Mr. T. Harayama for helpful discussions. The critical comments and suggestions given by the referees helped the authors to improve the presentation of the paper.

References

[1] P.K. Agarwal, M. de Berg, J. Matousek, O. Schwarzkopf, Constructing levels in arrangements and higher order Voronoi diagram, SIAM J. Comput. 27 (1998) 654–667.
[2] B. Chazelle, Lower bounds for orthogonal range searching, II: the arithmetic model J. ACM 37 (1990) 439–463.
[3] B. Chazelle, The Descripancy Method: Randomness and Complexity, Cambridge University Press, Cambridge, 2000.
[4] B. Chazelle, J. Friedman, A deterministic view of random sampling and its use in geometry, Combinatorica 10 (1990) 229–249.
[5] B. Chazelle, L.J. Guibas, Fractional cascading—II. Applications, Algorithmica 1 (1986) 163–191.
[6] R. Cole, C.K. Yap, Geometric retrieval problems, Proc. 24th IEEE Symp. on Foundation of Computer Science, 1983, pp. 112–121.
[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.
[8] H. Edelsbrunner, Algorithms in Combinatorial Geometry, Springer, Berlin, 1987.
[9] H. Edelsbrunner, L.J. Guibas, J. Stolfi, Optimal point location in monotone subdivision, SIAM J. Comput. 15 (1986) 317–340.
[10] H. Everett, J.-M. Robert, M. van Kreveld, An optimal algorithm for the ($\leqslant k$)-levels, with applications to separation and transversal problems, Int. J. Comput. Geom. Appl. 6 (1996) 247–261.
[11] J. Hershberger, Finding the upper envelope of n line segments in $O(n\log n)$ time, Inform. Process. Lett. 33 (1989) 169–174.
[12] D.T. Lee, Y.T. Ching, The power of geometric duality revisited, Inform. Process. Lett. 21 (1985) 117–122.
[13] J. Matoušek, Range searching with efficient hierarchical cutting, Discrete Comput. Geom. 10 (1993) 157–182.
[14] P. Mitra, B.B. Chaudhuri, Efficiently computing the closest point to a query line, Pattern Recognition Lett. 19 (1998) 1027–1035.
[15] K. Mulmuley, Computational Geometry: An Introduction through Randomized Algorithms, Prentice-Hall, Englewood Cliffs, NJ.
[16] J. Pach, P.K. Agarwal, Combinatorial Geometry, Wiley, Inc., New York, 1995.
[17] M. Sharir, P.K. Agarwal, Davenport-Schinzel Sequence and their Geometric Applications, Cambridge University Press, Cambridge, 1995.
[18] C.-S. Shin, S.Y. Shin, K.-Y. Chwa, The widest k-dense corridor problems, Inform. Process. Lett. 68 (1998) 25–31.