Selection in Monotone Matrices and Computing $k$th Nearest Neighbors

Pankaj K. Agarwal

Department of Computer Science, Duke University, P.O. Box 99129, Durham, North Carolina 27708-9129.

and

Sanjeev Sen

Department of Computer Science, Indian Institute of Technology, New Delhi, India

Received March 8, 1994

An $m \times n$ matrix $A = (a_{i,j})$, $1 \leq i \leq m$ and $1 \leq j \leq n$, is called a totally monotone matrix if for all $i_1, i_2, j_1, j_2$, satisfying $1 \leq i_1 < i_2 \leq m$, $1 \leq j_1 < j_2 \leq n$,

$$a_{i_1, j_1} < a_{i_1, j_2} \Rightarrow a_{i_2, j_1} < a_{i_2, j_2}.$$

We present an $O(m + n \sqrt{n} \log n)$ time algorithm to select the $k$th smallest item from an $m \times n$ totally monotone matrix for any $k \leq m$. This is the first sub-quadratic algorithm for selecting an item from a totally monotone matrix. Our method also yields an algorithm of the same time complexity for a generalised row-selection problem in monotone matrices. Given a set $S = \{p_1, \ldots, p_n\}$ of $n$ points in convex position and a vector $k = (k_1, \ldots, k_n)$, we also present an $O(n^{1/2} \log^c n)$ algorithm to compute the $k_i$th nearest neighbor of $p_i$, for every $i \leq n$, here $c$ is an appropriate constant. This algorithm is considerably faster than the one based on a row-selection algorithm for monotone matrices. If the points of $S$ are arbitrary, then the $k_i$th nearest neighbor of $p_i$, for all $i \leq n$, can be computed in time $O(n^{7/4} \log^c n)$, which also improves upon the previously best-known result.
1. INTRODUCTION

An \( m \times n \) matrix \( \mathcal{A} = (a_{i,j}) \), \( 1 \leq i \leq m \) and \( 1 \leq j \leq n \), is called a totally monotone matrix if for all \( i_1, i_2, j_1, j_2 \), satisfying \( 1 \leq i_1 < i_2 \leq m \), \( 1 \leq j_1 < j_2 \leq n \),

\[
a_{i_1,j_1} < a_{i_2,j_2} \Rightarrow a_{i_1,j_2} < a_{i_2,j_1}.
\]  

(1)

See Fig. 1 for an example.

Totally monotone matrices were originally introduced by Aggarwal et al. [4]. They presented an \( O(m + n) \)-time algorithm for computing the leftmost maximal element in each row of an \( m \times n \) totally monotone matrix, and applied it to a number of problems in computational geometry and VLSI routing, including all-farthest neighbors of convex polygons. Since the publication of their paper, several new applications of monotone matrices have been discovered, e.g., dynamic programming, largest area (or perimeter) triangle in a set of points, economic lot problems, etc.; see [5, 6, 7, 8, 9, 22, 25, 29, 30, 31, 32, 38, 39] for some of these applications.

Most of the papers cited above, however, consider only the problem of computing the maximal or minimal elements of each row. Surprisingly, very little is known about more general selection problems for monotone matrices. There are two natural selection problems for monotone matrices.

(i) **Array selection**: Given an \( m \times n \) totally monotone matrix \( \mathcal{A} \), and a positive integer \( k \leq mn \), find the \( k \)th smallest element, \( x_k \), of \( \mathcal{A} \), i.e.,

\[
\left| \{ a_{i,j} \mid a_{i,j} < x_k, a_{i,j} \in \mathcal{A} \} \right| < k \text{ and } \left| \{ a_{i,j} \mid a_{i,j} \leq x_k, a_{i,j} \in \mathcal{A} \} \right| \geq k.
\]

(ii) **Row selection**: Given an \( m \times n \) totally monotone matrix \( \mathcal{A} \) and a positive integer \( k \leq n \), find the \( k \)th smallest element of each row of \( \mathcal{A} \).

\[
\begin{pmatrix}
59 & 47 & 20 & 37 & 7 & 11 & 2 \\
25 & 15 & 14 & 19 & 6 & 12 & 10 \\
40 & 33 & 24 & 34 & 22 & 28 & 24 \\
21 & 15 & 7 & 17 & 6 & 13 & 11 \\
32 & 28 & 23 & 37 & 32 & 44 & 45 \\
88 & 85 & 76 & 92 & 90 & 100 & 110 \\
31 & 30 & 34 & 53 & 51 & 66 & 73 \\
6 & 9 & 13 & 32 & 32 & 52 & 62 \\
18 & 29 & 35 & 43 & 55 & 78 & 95
\end{pmatrix}
\]

\( \mathcal{A} \)

**Fig. 1.** An example of a totally monotone matrix.
One can, of course, use the well-known linear-time selection algorithms [12, 37] to solve both of these problems in $O(mn)$ time, so the challenge is to develop an $o(mn)$-time algorithm by exploiting the properties of monotone matrices. Selecting the $k$th smallest element from a set possessing certain properties has been studied by several researchers [23, 24, 28]. Frederickson and Johnson [24] presented an $O(m \log(2n/m))$-time algorithm to select the $k$th smallest item from an $m \times n$ matrix each of whose rows and columns is sorted in nondecreasing order. Kravets and Park [30] showed that for an $m \times n$ totally monotone matrix whose transpose is also totally monotone, the $k$th smallest element can be computed in time $O(m + n + k \log(mn/k))$. Their algorithm is optimal for $k = O(1)$, but its running time is $\Omega(mn)$ when $k$ is large. Moreover, their algorithm does not work if $A^T$ is not totally monotone.

Kravets and Park also presented an $O(k(m + n))$-time algorithm for the row selection problem. Again, their problem is quite inefficient for large values of $k$. Recently, Mansour et al. [31] developed another algorithm whose running time is $O(\sqrt{nm} \log n + m)$. Both of these algorithms rely on the fact that the value of $k$ is the same for all rows of $A'$, so they do not extend to the following generalized row-selection problem: Given an $m \times n$ totally monotone matrix $A'$ and a vector $k = (k_1, \ldots, k_m)$ of length $m$, where $0 < k_i \leq n$, compute the $k_i$th smallest element of the $i$th row of $A'$, for all $1 \leq i \leq m$. Alon and Azar [10] proved that $O(n \sqrt{m} \log n \sqrt{\log n})$ comparisons are sufficient to solve the generalized row selection problem, but their approach falls short of giving an efficient algorithm.

In this paper we present an $O((m + n)\sqrt{n} \log n)$-time algorithm for the array-selection problem. This is the first subquadratic algorithm for the array-selection problem. Unlike the Kravets–Park algorithm, it does not require $A^T$ to be totally monotone. A natural variant of our algorithm solves the generalized row-selection problem in time $O((m + n)\sqrt{n} \log n)$.

In the second part of this paper, we consider the all $k$th nearest-neighbor problem for a convex set: Given a set $S$ of $n$ points in the plane in convex position and a parameter $k \leq n$, find the $k$th nearest neighbor of every point in $S$. As shown in [30, 31], this problem can be reduced to the row-selection problem. However, there are known algorithms that solve the all $k$th nearest-neighbor problem for an arbitrary set of points in the plane directly. Using $k$th order Voronoi diagrams, the $k$th nearest neigh-

\[1\] In order to obtain an $o(mn)$ algorithm, we have to assume that $A'$ is not represented explicitly. We assume, that any particular entry $a_{ij}$ can be computed in constant time. For instance, given a set ($p_1, \ldots, p_n$) of points, $a_{ij}$ may be the distance between $p_i$ and $p_j$. We do not compute all pairwise distances explicitly, but any of them can be computed in $O(1)$ time. If the time required to compute $a_{ij}$, for a given pair $i, j$, is $\Theta(mn)$, then we have to multiply the running time of the algorithm by a factor of $O(mn)$. 


The running time has been improved by Vaidya [40] to $O(nk \log n)$, and recently by Callahan and Kosaraju [13] to $O(n \log n + nk)$ (see also [18]). A drawback of these algorithms is that their running time depends on $k$, and is $\Omega(n^2)$ for $k = \Omega(n)$. Agarwal and Matoušek presented at $O(n^{3/2 + \epsilon})$ time algorithm for the all $k$th nearest neighbor problem, for any $\epsilon > 0$ [12].

Here, we present an $O(n^{4/3} \log^c n)$-time algorithm, for some constant $c > 0$, for the all $k$th nearest-neighbor problem for the special case when the points of $S$ are in convex position. In fact, we present an algorithm for a more general problem. We show that an arbitrary set $S$ of $n$ points and a convex polygon $P$, with $n^{O(1)}$ vertices, in the plane can be preprocessed in time $O(n^2 \log n)$ into a data structure of size $O(n^2)$ so that for a query point $p$ lying on $\partial P$ (the boundary of $P$), the $k$th nearest neighbor of $p$ can be computed in time $O(\log n)$. This data structure can also count the number of points of $S$ lying in a disk whose center lies on $P$. (If the center of a query disk lies anywhere in the plane, then the best known data structure with $O(\log n)$ query time requires $O(n^3)$ space [14].)

Combining this data structure with the range-searching data structure described in [3], we can preprocess $S$ into a data structure of size $O(s)$, for any $s$ ($n \leq s \leq n^2$), so that a $k$th nearest neighbor query as described above, can be answered in time $O((n/\sqrt{s}) \log^c n)$ time, where $c > 1$ is some constant. By setting $s = n^{4/3}$ and querying the data structure with each point of $S$, the all $k$th nearest neighbor problem for a convex set can be solved in time $O(n^{4/3} \log^c n)$. For an arbitrary set of points in the plane, the running time is $O(n^{7/6} \log^c n)$.

2. SELECTION IN MONOTONE MATRICES

In this section we present algorithms for the array-selection and the generalized row selection problems. Let $\mathcal{A} = (a_{ij})$, $1 \leq i \leq m$, $1 \leq j \leq n$, be an $m \times n$ totally monotone matrix, and let $k \leq mn$ be a positive integer. The algorithm works in two steps. In the first step, it partitions $\mathcal{A}$ into a family $\mathcal{S} = \{L_1, \ldots, L_t\}$ of lists so that each list $L_i \in \mathcal{S}$ satisfies the following two properties.

(i) All elements of $L_i$ belong to the same row, say $r_i$.

(ii) The elements of each $L_i$ are in a nondecreasing order.

---

2 Here we are assuming that the vertices of $P$ are given as an array, sorted in a clockwise direction, so that for any integer $t$, the $i$th vertex of $P$ can be accessed in $O(1)$ time.
The lists of \( \mathcal{L} \) will be stored implicitly, as described below. After having computed the family of lists \( \mathcal{L} \), we select the \( k \)th smallest element among the elements of lists of \( \mathcal{L} \) in time \( O(|\mathcal{L}| \log n) = O(n \sqrt{n} \log n) \), using a simplified version of the Frederickson–Johnson algorithm [24]. We now describe each of the two steps in detail.

2.1. Computing the Family of Lists

We first describe the algorithm for computing the family \( \mathcal{L} \). In addition to \( \mathcal{L} \), we also compute a collection \( \Pi = \{\pi_1, \ldots, \pi_r\} \) of arrays, referred to as index-arrays, which will be used to represent \( \mathcal{L} \) implicitly. Let \( a_{i,k} \) be the \( k \)th item of \( L_i \). For each \( L_i \), we store the value of \( r_i \) and a pointer to an array \( \pi_i \). The array \( \pi_j \) has the same size as \( L_i \), and that the \( k \)th item of \( \pi_i \) is \( k \) if the \( k \)th item of \( L_i \) is \( a_{i,k} \); we will denote \( \pi_j \) as \( \varphi(L_i) \).

Properties of Monotone Matrices. We will use the following properties of monotone matrices, which are quite easy to prove; see, e.g., [30, 31] for proofs.

**Lemma 2.1.** If \( \mathcal{A} \) is a totally monotone matrix, then any submatrix of \( \mathcal{A} \) is also a totally monotone matrix.

**Lemma 2.2.** For any three integers \( i, j, k \), satisfying \( 1 \leq i \leq m, 1 \leq j < k \leq n \),

\[
(i) \quad a_{i,j} \geq a_{i,k} = a_{i,j} \geq a_{i,k} \quad \text{for all} \quad i \leq i, \text{ and}
(ii) \quad a_{i,j} \leq a_{i,k} = a_{i,j} \leq a_{i,k} \quad \text{for all} \quad i \geq i.
\]

We also use a well-known result of Erdős and Szekeres [21], which has been applied to a number of other geometric problems; see [33, 11]. Let \( U = \langle u_1, u_2, \ldots, u_m \rangle \) be a sequence of real numbers. A subsequence \( U' = \langle u_{i_1}, u_{i_2}, \ldots, u_{i_t} \rangle \) of \( U \) is called monotone if either \( u_{i_1} \leq u_{i_2} \leq \cdots \leq u_{i_t} \) or \( u_{i_1} \geq u_{i_2} \geq \cdots \geq u_{i_t} \).

**Lemma 2.3 (Erdős–Szekeres).** Given a sequence \( U \) of \( n \) real numbers, there always exists a monotone subsequence \( U' \) of length at least \( \lceil \sqrt{n} \rceil \).

Dijkstra [19] gave a simple \( O(n \log n) \)-time algorithm for computing a longest monotone subsequence of \( U \). Recently, Bar Yehuda and Fogel [11] presented an \( O(n^{3/2}) \)-time algorithm for partitioning \( U \) into a collection of at most \( 2\sqrt{n} \) monotone subsequences.

**Sublist Algorithm.** We now present a divide-and-conquer algorithm for computing \( \mathcal{L} \). At each recursive step of the algorithm we have a \( u \times v \) submatrix \( M \) of \( \mathcal{A} \) (see Fig. 2); initially \( M = \mathcal{A} \). \( M \) consists of a contiguous block of \( u \) rows of \( \mathcal{A} \), say \( \alpha + 1, \ldots, \alpha + u \), and a subsequence \( C(M) \) of columns of \( \mathcal{A} \), i.e., \( C(M) = \{a_{i,j} \mid \alpha + 1 \leq i \leq \alpha + u, j \in C(M) \} \). \( C(M) \) is repre-
\[ M = \begin{array}{ccccccccc}
59 & 47 & 20 & 37 & 7 & 11 & 2 \\
25 & 15 & 14 & 19 & 6 & 12 & 10 \\
40 & 33 & 24 & 34 & 22 & 28 & 24 \\
21 & 15 & 7 & 17 & 6 & 13 & 11 \\
\end{array}
\]

\[ L_1 = \langle 32, 37, 44, 45 \rangle \]
\[ L_2 = \langle 32, 37, 44, 45 \rangle \]
\[ L_3 = \langle 31, 33, 34, 35 \rangle \]
\[ L_4 = \langle 6, 32, 52, 62 \rangle \]
\[ L_5 = \langle 18, 43, 78, 95 \rangle \]

\[ M_1 = \begin{array}{ccccccccc}
59 & 47 & 20 & 37 & 7 & 11 & 2 \\
25 & 15 & 14 & 19 & 6 & 12 & 10 \\
40 & 33 & 24 & 34 & 22 & 28 & 24 \\
21 & 15 & 7 & 17 & 6 & 13 & 11 \\
\end{array}
\]

Fig. 2. Decomposing \( M \) into two \( M_1, M_2 \): \( \sigma = \langle 32, 37, 44, 45 \rangle \), \( \pi_1 = \langle 1, 4, 6, 7 \rangle \).

presented by specifying the values of \( \alpha \) and \( \nu \) and the subsequence \( C(M) \). By Lemma 2.1, \( M \) is totally monotone matrix. Let \( \mathcal{S}' = \langle L_1, \ldots, L_{\nu} \rangle \) and \( \Pi' = \langle \pi_1, \ldots, \pi_{\nu} \rangle \) be the families of lists and index-arrays, respectively, that we have computed so far.

If \( M \) consists of a single column, i.e., \( C(M) = \langle z \rangle \), then we set

\[ \Pi' = \Pi' \cup \{ \pi_{\nu+1} \} \quad \text{and} \quad \mathcal{S}' = \mathcal{S}' \cup \{ L_{\nu+1}, \ldots, L_{\nu+u} \}. \]

where \( \pi_{\nu+1} = \langle z \rangle \) and, for \( 1 \leq i \leq \nu \), \( L_{\nu+1} = \langle a_{i \alpha+1 \nu}, \bar{r}_{\nu+1} = \alpha + \bar{t}, \text{and} \quad \varphi(L_{\nu+1}) = \pi_{\nu+1} \).

Next, assume that \( \nu > 1 \) (\( M \) consists of more than one column). Set \( w = \lceil u/2 \rceil \), and let

\[ U = \langle a_{\alpha + w, j} \mid j \in C(M) \rangle \]

be the \( w \)-th row of \( M \). Using Dijkstra's algorithm, we compute the longest monotone subsequence

\[ \sigma = \langle a_{\alpha + w, j_1}, a_{\alpha + w, j_2}, \ldots, a_{\alpha + w, j_k} \rangle \]

of \( U \). There are two cases to consider depending on whether \( \sigma \) is nonincreasing or nondecreasing.

Case A. Suppose \( \sigma \) is a monotonic nonincreasing subsequence, then for all \( x, \alpha + 1 \leq x \leq \alpha + w \), the subsequence

\[ \langle a_{x, j_1}, \ldots, a_{x, j_k} \rangle \]
is also monotonic nonincreasing (cf. Lemma 2.2). We set

$$\Pi = \Pi' \cup \{ \pi_{i+1} \} \quad \text{and} \quad \mathcal{S}'' = \mathcal{S}' \cup \{ L_{i+1}, \ldots, L_{i+w} \}. \quad (1)$$

where

$$\pi_{i+1} = \langle j_k, j_{k-1}, \ldots, j_1 \rangle \quad (2)$$

$$L_{i+1} = \langle a_{(\alpha + \gamma + i), j_k}, a_{(\alpha + \gamma + i, j_{k-1}), \ldots, a_{(\alpha + \gamma + i, j_1)} \rangle. \quad (3)$$

$r_{i+1} = \alpha + i$ and $\varphi(L_{i+1}) = \pi_{i+1}$ for $1 \leq i \leq w$. As mentioned in the introduction, we store $L_{i+1}$ implicitly by storing the value of $r_{i+1}$ and a pointer to $\pi_{i+1}$.

We represent the remaining entries of $M$, the ones that do not belong to any list $L_i$, into two matrices $M_1, M_2$, and recursively decompose them into sorted lists. $M_1 = (a_{i,j})$ is composed of the bottom $u - w$ rows of $M$ and all the columns of $\mathcal{S}'$, i.e., $\alpha + w + 1 \leq i \leq \alpha + u$, and $C(M_1) = C(M)$. $M_2 = (a_{i,j})$ is composed of the top $w$ rows of $M$ and those columns of $M$ that are not in $\sigma$, i.e., $\alpha + 1 \leq i \leq \alpha + w$, and $C(M_2) = C(M) - \langle j_1, \ldots, j_k \rangle$.

Case B. If $\sigma$ is a monotonically nondecreasing subsequence, then for all $x, \alpha + w \leq x \leq \alpha + u$, the subsequence $\langle a_{x,j_k}, \ldots, a_{x,j_1} \rangle$ is also monotonic non-decreasing, so we set

$$\Pi = \Pi' \cup \{ \pi_{i+1} \} \quad \text{and} \quad \mathcal{S}'' = \mathcal{S}' \cup \{ L_{i+1}, \ldots, L_{i+w-1} \}. \quad (4)$$

where

$$\pi_{i+1} = \langle j_k, j_{k-1}, \ldots, j_1 \rangle \quad (5)$$

$$L_{i+1} = \langle a_{(\alpha + w + i), j_k}, a_{(\alpha + w + i, j_{k-1}), \ldots, a_{(\alpha + w + i, j_1)} \rangle. \quad (6)$$

$r_{i+1} = \alpha + w + i$, and $\varphi(L_{i+1}) = \pi_{i+1}$ for $0 \leq i \leq u - w$. $M_1 = (a_{i,j})$ is composed of the top $w - 1$ rows of $M$ and all the columns of $\mathcal{S}'$, i.e., $\alpha + 1 \leq i \leq \alpha + w$ and $C(M_1) = C(M)$. $M_2$ is composed of the bottom $u - w + 1$ rows of $M$ and those columns of $M$ that are not in $\sigma$, i.e., $\alpha + w \leq i \leq \alpha + u$ and $C(M_2) = C(M) - \langle j_1, \ldots, j_k \rangle$; see Fig. 2.

Next, let us analyze the size of $\mathcal{S}'$ and the total time spent by the algorithm. Let $\psi(u, v)$ be the maximum number of lists generated by the algorithm. By Lemma 2.3, the longest subsequence has length at least $\lceil \sqrt{v} \rceil$, so we get the following recurrence

$$\psi(u, v) \leq \begin{cases} u & \text{if } v = 1, \\ \psi\left(\left\lfloor \frac{u}{2} \right\rfloor, v\right) + \psi\left(\left\lceil \frac{u}{2} \right\rceil, v - \left\lfloor \sqrt{v} \right\rfloor\right) & \text{if } v > 1. \end{cases}$$
We claim that the solution of this recurrence is
\[ \psi(u, v) \leq 2u\sqrt{v}. \tag{7} \]

Indeed, (7) is obviously true for \( v = 1 \). For \( v > 1 \), by induction hypothesis, we have
\[
\psi(u, v) \leq \psi\left(\left\lfloor \frac{u}{2} \right\rfloor, v\right) + \psi\left(\left\lfloor \frac{u}{2} \right\rfloor, v - \sqrt{v}\right) + \left\lfloor \frac{u}{2} \right\rfloor
\]
\[
\leq 2\left\lfloor \frac{u}{2} \right\rfloor \sqrt{v} + 2\left\lfloor \frac{u}{2} \right\rfloor \sqrt{v - \sqrt{v}} + \left\lfloor \frac{u}{2} \right\rfloor
\]
\[
\leq u\left(\sqrt{v} + \sqrt{v - \sqrt{v}} + \frac{1}{2}\right)
\]
\[
\leq 2uv^{1/2},
\]
because \( \sqrt{v - \sqrt{v}} + 1/2 < \sqrt{v} \).

Let \( T(u, v) \) be the maximum running time of the algorithm. Since we spend \( O(v, \log v) \) time in computing the monotonic subsequence \( \sigma \) and \( O(u + v) \) time in generating the lists plus the two submatrices \( M_1 \) and \( M_2 \), we have
\[
T(u, v) \leq T\left(\left\lfloor \frac{u}{2} \right\rfloor, v\right) + T\left(\left\lfloor \frac{u}{2} \right\rfloor, v - \sqrt{v}\right) + O(u + v \log v).
\]

Following the same calculations as above, it can be shown that
\[
T(u, v) = O(uv^{1/2} + v^{3/2} \log u \log v).
\]
This implies that \( |\mathcal{L}| \leq 2m\sqrt{n} \), and that the total time spent in computing \( \mathcal{L} \) is \( O(m\sqrt{n} + n^{3/2} \log n \log m) \).

The running time can be improved to \( O(m\sqrt{n} + n^{3/2} \log \log n) \) using the algorithm of Bar Yehuda and Fogel [11]. Instead of computing the longest monotone subsequence of \( U \), we partition \( U \) into at most \( 2\sqrt{v} \) monotone sequences, \( U_1, \ldots, U_r \), each of length at most \( \sqrt{v} \). Suppose \( U_j \) is a monotonic nonincreasing sequence. We update \( \Pi', \mathcal{L}' \) as in (1)–(3). Let \( M_j \) be the submatrix of \( M \) consisting of bottom \( \lfloor u/2 \rfloor \) rows of \( M \) and those columns of \( M \) that are in \( U_j \). Similarly, if \( U_j \) is a monotonic nondecreasing subsequence, we update \( \mathcal{L}', \Pi \) as in (4)–(6), and set \( M_j \) to be the submatrix of \( M \) consisting of top \( w - 1 \) rows of \( M \) and those columns of \( M \) that are in \( U_j \). We recursively decompose \( M_j \) into sorted lists. Repeating this procedure for all \( U_j \), \( 1 \leq j \leq r \), we obtain the family \( \mathcal{L} \). The recurrence for the maximum running time \( T(u, v) \) now becomes
\[
T(u, v) \leq \sum_{j=1}^{r} T(\lfloor u/2 \rfloor, v_j) + O(uv^{1/2} + v^{3/2}).
\]
where \( \max_{1 \leq i \leq v} i \leq v \) and \( \sum_{i=1}^v \psi_i = v \). The solution of the above recurrence is easily seen to be

\[
T(u, v) = O(u \sqrt{v} + v^{3/2} \log \log v).
\]

Hence, we can conclude:

**Lemma 2.4.** Any \( m \times n \) totally monotone matrix can be decomposed in time \( O(m\sqrt{n} + n^{3/2} \log \log n) \) into a set \( L = \{ L_1, \ldots, L_t \} \) of \( O(m\sqrt{n}) \) nondecreasing lists. \( L \) is stored implicitly so that for any pair \( (i, j) \), the \( j \)th element of \( L_i \) can be accessed in \( O(1) \) time.

2.2. Selecting the \( k \)th Element

Let \( L = \{ L_1, \ldots, L_t \} \) be a set of \( t \) lists, each sorted in nondecreasing order, and let us assume that, for any \( i \leq t \) and \( j \leq |L_i| \), the \( j \)th item of \( L_i \) can be obtained in \( O(1) \) time. We present an algorithm, which is a simplified version of the algorithm by Frederickson and Johnson [24], for selecting the \( k \)th smallest item in \( \bigcup_{i=1}^t L_i \).

Let \( p = \lceil \log_2 (\max_{1 \leq i \leq t} |L_i|) \rceil = \lceil \log_2 n \rceil \). Without loss of generality assume that each list \( L_i \in L \) has exactly \( 2^p \) elements; if \( |L_i| < 2^p \), then we assume that \( 2^p - |L_i| \) \( \infty \)'s are padded at the end of \( L_i \). The algorithm works in \( p \) phases. In the beginning of the \( i \)th phase, we have a set \( \Lambda \) of at most \( 2^i \) lists, each of size \( m_i = 2^{p-i} \) (including the padded \( \infty \)'s), and a parameter \( s \leq k \). Each list in \( \Lambda \) is a contiguous sublist of some list in \( L \), and the \( s \)th element of \( \bigcup \Lambda \) is the same as the \( k \)th element of \( \bigcup L \).

Initially, \( \Lambda = L \) and \( s = k \). The goal is to find the \( s \)th element in the set \( \bigcup \Lambda \).

In the \( i \)th set we first divide each list \( \lambda \in \Lambda \) into two lists \( \lambda_1, \lambda_2 \); \( \lambda_1 \) consists of the first \( m_i/2 \) elements of \( \lambda \), and \( \lambda_2 \) consists of the next \( m_i/2 \) elements of \( \lambda \). Let \( \bar{\Lambda} \) denote the set of lists after the divide step. Next, we discard some of the lists in \( \bar{\Lambda} \) that are guaranteed not to contain the \( s \)th element of \( \bigcup \bar{\Lambda} \).

Recall that we are not storing the lists of \( \Lambda \) explicitly, so the divide step is somewhat involved. For each list \( \lambda \in \Lambda \), we store the following information:

(i) The list \( L_j \in L \) that contains \( \lambda \), and

(ii) the index of the first item of \( \lambda \) in \( L_j \).

When we split \( \lambda \) into \( \lambda_1, \lambda_2 \), this information for both \( \lambda_1, \lambda_2 \) can be computed in \( O(1) \) time. The prune-step works in two stages. Let \( \sigma = \lfloor s/m_i \rfloor + i \), and let \( \lambda(i) \) denote the \( i \)th element of \( \lambda \). We select the \( \sigma \)th smallest element, \( x_\sigma \), from the set \( A = \{ \lambda(i) \mid \lambda \in \Lambda \} \). \( A \) can be obtained
in $O(|\overline{A}|)$ time and $x_\alpha$ can be computed in another $O(|\overline{A}|)$ time. If $\alpha < |\overline{A}|$, we discard all $|\overline{A}| - \alpha$ lists of $\overline{A}$ whose minimum item is not smaller than $x_\alpha$. Next, let $\beta = \lfloor s/n \rfloor - t$. If $\beta > 0$, we select the $\beta$th element, $x_{\beta}$, from the set $B = \{\lambda m_j \mid \lambda \in \overline{A}\}$, and discard all the lists whose maximum element is not larger than $x_{\beta}$. The second stage prunes at least $\beta$ lists. Set $s = s - \beta m_j$.

After the $p$th step each list in $\overline{A}$ has at most one item. We then use the linear-time selection algorithm to find the $s$th item in another $O(t)$ time.

Notice that after each phase, at most $2t$ lists are retained in $\overline{A}$, because at least $|\overline{A}| - \alpha + \beta \leq |\overline{A}| - 2t$ lists are deleted by the prune step. The running time of the algorithm is thus $O(p) = O(m\sqrt{n} \log n)$. The correctness of the algorithm follows from the fact that in the $i$th step, the $s$th element of $\cup \overline{A}, x_s$, cannot lie in one of the discarded lists; see the original paper [24] for a proof.

Plugging this algorithm to Lemma 2.4, we can conclude:

**Theorem 2.5.** Let $A$ be an $m \times n$ totally monotone matrix, and let $k \leq mn$ be some positive integer. The $k$th smallest item of $A$ can be computed in time $O(m\sqrt{n} \log n + n^{3/2}\log \log n)$.

### 2.3. Generalized Row Selection

By modifying the above algorithm in an obvious manner, we can solve the generalized row selection problem: Let $k = \langle k_1, k_2, \ldots, k_m \rangle$ be a vector of length $m$, where each $k_i \leq n$ is a positive integer. The goal is to select the $k_i$th smallest item from the $i$th row of $A$. We decompose $A$ into a set $P$ of nondecreasing lists as above. $P$ is again represented implicitly using the set $\Pi$ of index-arrays. Let

$$P_i = \{L_j \mid r_i = j\}$$

be the subset of lists belonging to the $i$th row of $A$. We run the above selection algorithm for each $P_i$ separately and select the $k_i$th item of $P_i$. The running time for each $P_i$ is $O(|P_i| \log n)$, thereby implying that the overall running time is again $O(m\sqrt{n} \log n + n^{3/2}\log \log n)$. Hence, we have

**Theorem 2.6.** Let $A$ be an $m \times n$ totally monotone matrix and let $k$ be a vector of length $m$ as defined above. The $k_i$th smallest item of the $i$th row of $A$, for all $1 \leq i \leq n$, can be computed in time $O(m\sqrt{n} \log n + n^{3/2}\log \log n)$.

We conclude this section by mentioning an application of Theorems 2.5 and 2.6. Let $P$ be a simple $n$-gon. The geodesic distance between two points $p, q \in P$ is the length of the shortest path connecting $p, q$ that lies
inside \( P \). As shown in [27], the geodesic-distance matrix for the vertices of \( P \) can be represented as a totally monotone matrix. Moreover, using the data structure of Guibas and Hershberger [26], each entry of the matrix can be computed in \( O(\log n) \) time. Hence, the \( k \)th smallest geodesic distance between the vertices of \( P \) can be computed in time \( O(n^{3/2}\log^2 n) \) time, and the \( k \)th closest neighbor of every vertex of \( P \) can also be computed in time \( O(n^{3/2}\log^2 n) \).

**Theorem 2.7.** Let \( P \) be a simple \( n \)-gon. For a given positive integer \( k \leq \left( \frac{n}{2} \right) \), the \( k \)th smallest geodesic distance between the vertices of \( P \) can be computed in time \( O(n^{3/2}\log^2 n) \). For a given vector \( \mathbf{k} = \langle k_1, \ldots, k_n \rangle \), where each \( k_j \leq n \), the \( k \)th nearest neighbor of the \( i \)th vertex of \( P \) (over all \( 1 \leq i \leq n \)) can be computed in time \( O(n^{3/2}\log^2 n) \).

**Remark 2.8.** It might be possible to improve the running time in the above theorem by a logarithmic factor, using the ideas similar to those in [27], but we feel that this improvement is not worth the effort.

3. COMPUTING ALL \( k \)th NEAREST NEIGHBORS

Let \( S \) be a set of \( n \) points in the plane in convex position and let \( \mathbf{k} = \langle k_1, k_2, \ldots, k_n \rangle \) be a vector of length \( n \), where each \( k_j \leq n \) is a positive integer. The goal is to compute the \( k \)th nearest neighbor of \( p_i \) for \( 1 \leq i \leq n \). Before describing the algorithm in detail, we explain our overall approach.

We first present a data structure for the circular range-searching problem: Let \( S \) be a set of \( n \) points and \( P \) a convex polygon, with \( n^{O(1)} \) vertices, in the plane. Preprocess \( S \) and \( P \) into a data structure so that for a query disk \( D \) whose center lies on \( \partial P \), \( |D \cap S| \) can be computed efficiently. Using this algorithm as a subroutine and applying the parametric-search technique, we obtain an efficient algorithm for answering \( k \)th nearest-neighbor queries for points on \( \partial P \) (i.e., for a query point \( p \in \partial P \) and a positive integer \( k \leq n \), we wish to determine the \( k \)th nearest neighbor of \( p \)).

3.1. **Circular Range Searching**

In this subsection we show that if the center of the query disk always lies on the boundary of a convex polygon \( P \), then a circular range query can be answered in \( O(\log n) \) time using only \( O(n^2) \) space. We then combine it with the data structure of Agarwal and Matoušek [3] to obtain a space/query-time trade-off.
**Data Structure with $O(\log n)$ Query Time.** Let $S$ be a set of $n$ points in the plane. For a pair of points $p_i, p_j \in S$, let $\ell_{ij}$ denote the perpendicular bisector of $p_i$ and $p_j$, and let

$$\mathcal{L} = \{\ell_{ij} | 1 \leq i < j \leq n\}.$$ 

For a point $q$ in the plane, let $\sigma(q)$ denote the sequence of points of $S$ sorted in a nondecreasing order of their distances from $q$, i.e., if $d(q, p_i) \leq d(q, p_j) \leq \cdots \leq d(q, p_n)$, then

$$\sigma(q) = \langle p_{i_1}, p_{i_2}, \ldots, p_{i_n} \rangle.$$ 

We will refer to $\sigma(q)$ as the distance-ordering of $S$ with respect to $q$. It can be easily verified that, for any two points $q, q'$ lying in the same face of $A(\mathcal{L})$, $\sigma(q) = \sigma(q')$. Abusing the notation slightly, let $\sigma(f)$ denote the distance-ordering with respect to any point in the face $f$ of $A(\mathcal{L})$.

**Lemma 3.1.** If $f$ and $f'$ are two adjacent faces in $A(\mathcal{L})$ separated by the time $\ell_{k,l}$, then $p_k, p_l$ are adjacent in $\sigma(f)$ and $\sigma(f')$. Moreover, if

$$\sigma(f) = \langle p_{i_1}, \ldots, p_{i_m}, p_k, p_l, p_{i_m+2}, \ldots, p_{i_n} \rangle,$$

then

$$\sigma(f') = \langle p_{i_1}, \ldots, p_{i_m}, p_l, p_k, p_{i_m+2}, \ldots, p_{i_n} \rangle.$$ 

If we know the face $f$ of $A(\mathcal{L})$ that contains the center of a disk $D$ and the distance ordering $\sigma(f)$, then $|S \cap D|$ can be computed by a binary search on $\sigma(f)$—find the first element in $\sigma(f)$ whose distance from the center of $D$ is more than the radius of $D$. However, we cannot afford to store the entire arrangement $A(\mathcal{L})$, because there are $\Omega(n^2)$ faces in $A(\mathcal{L})$. This is where we use the fact that the center of $D$ lies on $\partial P$.

Partition $\partial P$ into maximal connected intervals that do not intersect any line of $\mathcal{L}$, so that each interval lies within a face of $A(\mathcal{L})$. Let $I = \langle I_1, \ldots, I_m \rangle$ be a sequence of the resulting intervals sorted along $\partial P$ in clockwise direction; $m = O(n^2)$, for any line intersects $\partial P$ in at most two points. Since each interval $I_j$ lies within a single face of $A(\mathcal{L})$, the distance ordering of $S$ with respect to all points of $I_j$ remains the same, and let $\sigma(I_j)$ denote this ordering. See Fig. 3. We store each $\sigma(I_j)$ in a binary search tree $T_j$, whose $i$th leftmost node $v$ stores the index $i$ (which we refer to as the rank of $v$ in $T_j$), and the $i$th element of $\sigma(I_j)$. Instead of storing $i$, we can also store the number of nodes in the subtree rooted at $i$. For a query disk $D$, whose center lies on $\partial P$, $|S \cap D|$ can be computed in a straightforward manner by traversing a path of $T_j$. Since $\sigma(I_j)$ and $\sigma(I_{j+1})$ differ only in two adjacent positions, and $\sigma(I_{j+1})$ can be obtained from $\sigma(I_j)$ by swapping two adjacent elements, we can use a
\[ \sigma(I_1) = (1, 2, 4, 5, 3) \]
\[ \sigma(I_2) = (1, 2, 4, 3, 5) \]
\[ \sigma(I_3) = (2, 3, 1, 4, 5) \]

Fig. 3. \( S = \{p_1, p_2, p_3, p_4, p_5\} \), \( P \), and \( \mathcal{S} = \{I_1, \ldots, I_{13}\} \).

persistent data structure to store all \( T_i \)'s [36]. We cannot use the update procedures described in [36] directly, because we are storing the rank of \( v \) (or the number of nodes in the subtree rooted at \( v \)) at each node \( v \) of \( T_j \); see [36] a more detailed discussion. To circumvent this problem we use a variant of their data structure, tailored to our application. Since the insertions and deletions are not arbitrary in our case, the update procedures are somewhat simpler than the ones described in [36].

Recall that each node in the persistent data structure, described by Sarnak and Tarjan [36], has an extra pointer (other than the standard "left" and "right" pointers). Whenever we want to modify the information stored at any node \( v \), we make a new copy \( v' \) and \( v \) and store the updated information there. If \( v \) is the root of the tree, we are done. Otherwise, let \( w \) be the parent of \( v \). If the extra pointer of \( w \) is free, it now points to \( v' \). If the extra pointer of \( w \) is not free, \( w \), too, is copied. Thus, copying
propagates through successive ancestors until the root is copied or a node with the free extra pointer is reached: see [36] for details.

We now describe how to modify the update procedures to suit our application. Let \( \mathcal{D} \) denote the overall data structure; \( \mathcal{D} \) is a dag that implicitly stores all \( T_j \)'s in the sense that \( T_j \) is a subgraph of \( \mathcal{D} \), and that given a node \( v \in T_j \), we can find the children of \( v \) in \( T_j \) in \( O(1) \) time. We also maintain an array \( A \) of length \( m \) whose \( j \)th entry stores a pointer to a root of \( \mathcal{D} \) that corresponds to the root of \( T_j \). Each node \( v \in \mathcal{D} \) stores the following information.

(i) A point of \( S \), denoted as \( val(v) \);

(ii) an integer, called \( rank(v) \), which is defined below;

(iii) three pointers—\( left \), \( right \), and \( extra \). Initially, when \( v \) is created, \( extra(v) \) pointer is free. As the algorithm progresses, the extra pointer is assigned to one of its children;

(iv) An integer, denoted as \( ver(v) \), which is 0 if the extra pointer of \( v \) is free; \( ver(v) = j \) if the extra pointer was assigned while processing \( I_j \);

(v) a bit, denoted as \( \xi(v) \); \( \xi(v) = 0 \) if the extra pointer is assigned to the left child of \( v \), and \( \xi(v) = 1 \) if the extra pointer is assigned as the right child of \( v \).

\( \mathcal{D} \) is constructed incrementally by processing the intervals of \( \mathcal{S} \) one by one. Let \( \mathcal{D}_j \) denote the data structures storing \( T_1, \ldots, T_j \). \( \mathcal{D}_1 = T_1 \) is a minimum-height binary tree storing \( \sigma(I_1) \). The \( i \)th leftmost node \( v_i \) of \( \mathcal{D}_i \) stores the \( i \)th element of \( \sigma(I_1) \), \( rank(v_i) = i \), and \( ver(v) = 0 \). Roughly speaking, \( \mathcal{D}_{i+1} \) can be obtained from \( \mathcal{D}_i \) as follows: let \( \mathcal{S}'_k \) be the line separating \( I_k \) and \( I_{k+1} \), and let \( u, v \) be the nodes corresponding to \( T_{j_k} \) that store \( p_k, p_t \), respectively. We make new copies \( u', v' \) of \( u \) and \( v \), and store \( p_k \) at \( u' \) and \( p_t \) at \( v' \) (i.e., swap \( p_k \) and \( p_t \)), and propagate the copying of nodes to the ancestors of \( u \) and \( v \), as sketched above. See Fig. 4.

We now describe in detail how to construct \( \mathcal{D}_{j+1} \) from \( \mathcal{D}_j \). The nodes \( u, v \) can be located in \( O(\log n) \) time by searching \( T_j \) with \( p_k \) and \( p_t \), respectively, see below for details of the search procedure. Since \( p_k \) and \( p_t \) are adjacent in \( \sigma(I_j) \), either \( u \) is a descendant of \( v \), or vice-versa. Without loss of generality, assume that \( u \) is a descendant of \( v \). We create a new node \( x \) with \( val(x) = p_k \), \( rank(x) = rank(u) \), \( ver(x) = 0 \). If \( ver(u) = 0 \) (i.e., the extra pointer of \( u \) is free), then \( left(x) = left(u) \) and \( right(x) = right(u) \). Otherwise, if \( \xi(u) = 0 \) (resp. \( \xi(u) = 1 \)), then \( left(x) = extra(u) \), \( right(x) = right(u) \) (resp. \( left(x) = left(u) \), \( right(x) = extra(u) \)). Let \( z \) be the highest ancestor of \( u \) such that, for all nodes \( w \) on the path from \( z \) to the parent of \( u \), \( ver(w) > 0 \). For each such node \( w \), we create a new node \( w' \). If \( w = u \), then we set \( val(w') = p_k \), otherwise we set \( val(w') = val(w) \).
We also set \( \text{rank}(w') = \text{rank}(w) \) and \( \text{ver}(w') = 0 \). The left and right pointers of \( w' \) are assigned in the same way as of \( x \). If \( z \) is not the root of \( \mathcal{D} \), then let \( y \) be the parent of \( z \). If \( z \) is the left (resp. right) child of \( y \), then we set \( \text{extra}(y) = z \), \( \text{ver}(y) = j + 1 \), and \( \xi(y) = 0 \) (resp. \( \xi(y) = 1 \)).

Next, if \( v \) is an ancestor of \( z \), we repeat the same procedure for \( v \). This completes the description of the update procedure. Finally, we store a pointer to the root of \( T_{j+1} \) at \( A[j+1] \).

Let \( D \) be a query disk with center \( c \in aP \) and radius \( r \). We compute \( |D \cap S| \) as follows. We first find in \( O(\log n) \) time the interval \( I_j \in \mathcal{I} \) that contains \( c \). \( A[j] \) gives the pointer to the root \( z \) of \( T_j \). We search in \( \mathcal{D} \) with the triple \((c, r, j)\), starting from \( z \). Suppose we are at node \( v \). Let \( p_k \) be the point stored at \( v \). If \( d(c, p_k) = r \), we return \( \text{rank}(v) \). If \( d(c, p_k) > r \) (resp. \( d(c, p_k) < r \)), we descend to the left (resp. right) child of \( v \) in \( T_j \). If \( v \) does not have the right (resp. left) child, then we return \( \text{rank}(v) \) (resp. \( \text{rank}(v) - 1 \)). The left and right children of \( v \) in \( T_j \) can be computed in \( O(1) \) time, as follows. If \( 0 < \text{ver}(v) \leq j \) and \( \xi(v) = 0 \) (resp. \( \xi(v) = 1 \)), then the extra pointer of \( v \) points to the left (resp. right) child of \( v \) in \( T_j \), and its right (resp. left) pointer points to the right (resp. left) child of \( v \) in \( T_j \). Otherwise, the left and right pointers points to the children of \( v \) in \( T_j \). The total query time is thus \( O(\log n) \) time. Notice that the same procedure can also compute the \( k \)th nearest neighbor of a point \( p \in P \).
Theorem 3.2. Let \( S \) be a set of \( n \) points in the plane, and let \( P \) be a convex polygon. \( S \) and \( P \) can be preprocessed in time \( O(n^2 \log n) \) into a data structure of size \( O(n^2) \), so that the number of points lying in a query disk whose center lies on \( \partial P \) can be counted in \( O(\log n) \) time. The same data structure can also compute the \( k \)th nearest neighbor of a query point \( p \in P \) in time \( O(\log n) \).

Space/Query-Time Trade-Off. We can obtain a space/query-time tradeoff by combining Theorem 3.2 with a data structure proposed by Agarwal and Matoušek [3] for the range-searching problem. For circular range searching, their data structure consists of a partition tree \( T \) on \( S \), each of whose node \( v \) is associated with a subset \( S_v \subseteq S \) and a "pseudo-trapezoid" \( \tau_v \) such that \( S_v \subseteq \tau_v \); \( \tau_v \) has at most four sides—the left and right sides are vertical segments and the top and bottom sides are portions of quadratic curves (see [3] for details). The degree of each node is \( r \) for some constant \( r \). If \( w_1, \ldots, w_r \) are the children of \( v \), then \( \max_{1 \leq i \leq r} |S_{w_i}| \leq 2|S_v|/r \) and \( \bigcup_{i=1}^r S_{w_i} = S_v \), so the height of \( T \) is \( O(\log n) \). If \( v \) is a leaf, then \( |S_v| = O(1) \). We say that a circle \( C \) crosses a node \( v \) if the interior of \( \tau_v \) intersects \( C \) and \( \tau_v \not\subseteq C \). \( T \) has the property that every circle crosses at most \( O(\sqrt{r} \log^{3/2} r) \) children of any node of \( T \). At each leaf \( v \) we store \( S_v \) itself, and at each internal node \( v \in T \) we store \( |S_v| \) and \( \tau_v \).

Let \( s \) be some fixed parameter for \( n \leq s \leq n^2 \). Suppose we want to construct a data structure of size \( s \). Delete all those nodes of \( T \) whose parents are associated with less than \( [rs/n] \) points. Let \( T' \) denote the resulting tree. Each leaf of \( T' \) contains at most \( [rs/n] \) points. For each leaf \( w \) of \( T' \), we preprocess \( S_w \) using Theorem 3.2 and replace the leaf \( w \) with the resulting structure. Let \( \Psi \) denote the overall data structure. We will refer to \( T' \) as the "top-structure" and to the structures stored at the leaves of \( T' \) as the "bottom-structures." Let \( L \) be the set of leaves of \( T' \). Then for any \( v \in L \), \( |S_v| \leq [rs/n] \) and \( \sum_{v \in L} |S_v| \leq n \). The total size of the data structure is at most

\[
O(n) + \sum_{v \in L} |S_v|^2 = O(n) + O\left(\frac{s^2}{n^2}\right) \cdot O(n^2/s) = O(s).
\]

The preprocessing time is easily seen to be \( O(s \log n) \).

Let \( D \) be a query disk with center \( c \) and radius \( r \). We visit \( \Psi \) in a top-down fashion, starting from the root. We maintain a global variable \( \text{count} \), which is initially set to 0. At each step, we visit a node \( v \). If \( v \) is a leaf of \( T' \), we compute \( |D \cap S_v| \) using the algorithm of Theorem 3.2, and add this quantity to \( \text{count} \). Otherwise, we do the following. If \( \tau_v \cap D = \emptyset \), we ignore \( v \). If \( \tau_v \subseteq D \), we add \( |S_v| \) to \( \text{count} \) and do not visit any children of \( v \). Finally, we recursively visit all the children of \( v \). Let \( Q(m) \) be the maximum query time of the procedure in querying a subtree storing a set
of \( m \) points. Since \( \partial D \) crosses at most \( O(\sqrt{r \log^{3/2} r}) \) children of \( v \), we obtain the following recurrence:

\[
Q(n) \leq \begin{cases} 
O(\log n) & \text{if } n_0 \leq s/n, \\
O(\sqrt{r \log^{3/2} r}) \sum_{i=1} Q(n/r) + O(r) & \text{if } n_0 > s/n.
\end{cases}
\]

The solution of the above recurrence is known to be

\[
Q(n_0) = O\left( \frac{n^{1+\varepsilon}}{\sqrt{s}} \right),
\]

see, e.g., [16]. The query time can be improved to \( O((n/\sqrt{s}) \log^c n) \) for some constant \( c > 0 \) by choosing \( r = n^\delta \), for some sufficiently small \( \delta > 0 \).

We leave out the details from here. Hence, we can conclude

**Theorem 3.3.** Let \( S \) be a set of \( n \) points in the plane, let \( P \) be a convex polygon, and let \( s \) be a parameter, with \( n \leq s \leq n^2 \). \( S \) can be preprocessed in time \( O(s \log n) \) into a data structure of size \( O(s) \), so that the number of points lying a query disk whose center lies on \( \partial P \) can be counted in \( O((n/\sqrt{s}) \log^c n) \) time, for some constant \( c > 0 \).

As mentioned in the introduction, if the center of a query disk lies anywhere in the plane, then \( |D \cap S| \) can be computed in time \( O(\log n) \) using \( O(n^2) \) space [14]. Plugging this data structure, instead of Theorem 3.2, into \( T \), we obtain:

**Theorem 3.4.** Let \( S \) be a set of \( n \) points in the plane, and let \( s \) be a parameter, with \( n \leq s \leq n^2 \). \( S \) can be preprocessed in time \( O(s \log n) \) into a data structure of size \( O(s) \), so that the number of points lying a query disk can be counted in \( O((n^{3/4}/s^{1/4}) \log^c n) \) time, for some constant \( c > 0 \).

**Remark 3.5.** Observe that the above query procedures can be modified, without affecting the query time, so that one can also detect whether a point of \( S \) lies on the boundary of the query disk.

### 3.2. Applying Parametric Searching

We now apply the parametric-searching technique, due to Megiddo [34, 35], to the above data structure for answering queries of the following form: Given a point \( q \) on the boundary of the convex polygon \( P \) and a parameter \( k \leq n \), determine its \( k \)th nearest neighbor, \( \varphi_k(q) \), in \( S \). The basic idea is the same as described in [2], but we have to modify their technique a little.
Let \( r_k = d(q, \varphi_k(q)) \), and let \( D_k \) be the disk of radius \( r_k \) centered at \( q \). We will query the data structure with \( D_k \). We, of course, do not know the value of \( r_k \), so we will simulate the query answering procedure, described above, without knowing the exact value of \( r_k \).

We search through \( \Psi \) level-by-level. Let \( V' \) be the set of nodes that we visit in the \( i \)th step. If a node \( v \in V' \) belongs to the bottom structure, we need to determine the relation between \( r_k \) and \( d(q, \varphi_k(v)) \) in order to determine whether the left or the right child of \( v \) has to be visited in the next step; let \( \delta_k = d(q, \varphi_k(v)) \). If \( v \) is a node of the top structure, then we want to determine whether \( D_k \) crosses \( \tau_o \), \( D_k \) contains \( \tau_o \), or \( D_k \) is disjoint from \( \tau_o \). Let

\[
\delta^+ = \min_{p \in r_k} d(q, p) \quad \text{and} \quad \delta^- = \max_{p \in r_k} d(q, p)
\]

(see Fig. 5): \( D_k \) crosses \( \tau_o \) if and only if \( \delta^- < r_k < \delta^+ \), and \( D_k \) contains \( \tau_o \) if and only if \( \delta^- < r_k \). The numbers \( \delta^- \), \( \delta^+ \) can be computed in \( O(1) \) time.

Let

\[
R_i = \{\delta^-, \delta^+ \mid v \in V' \text{ and } v \text{ is a node of the top structure} \}
\]

\[
\cup \{\delta^- \mid v \in V' \text{ and } v \text{ is a node of a bottom structure} \}
\]

We sort \( R_i \) and, by a binary search on the sorted list, we compute \( \rho_i \), the largest element of \( R_i \) that is less than or equal to \( r_k \). Each step of the binary search computes \( |D(q, r) \cap S| \) for some \( r \in R_i \). If \( |D(q, r) \cap S| < k \), then \( r < r_k \). If \( |D(q, r) \cap S| = k \) and \( \partial D \) contains a point of \( S \), then \( r = r_k \). Otherwise \( r > r_k \). By Theorem 3.3, each step of the binary search requires \( O(\sqrt{n} \log n) \) time. If \( \rho_i = r_k \), we already know the value of \( r_k \), so we stop right away. Next, assume that \( \rho_i < r_k \). For a node \( v \in V' \) of the bottom structure, if \( \delta^- < \rho_i \) (resp. \( \delta^+ > \rho_i \)), i.e., \( \delta^- < r_k \) (resp. \( \delta^+ > r_k \)), then we visit the right (resp. left) child of \( v \) in the next step. For a node

\[
\text{Fig. 5. } q, \delta^-, \delta^+.
\]
v \in V_i$ of the top structure, if $\delta_v^- > r_i$ (i.e., $\delta_v^- > r_i$), then $\tau_v$ and $D_k$ are disjoint, so we ignore $v$. If $\delta_v^- \leq r_i$ (i.e., $\delta_v^- < r_i$), then $\tau_v \subseteq D_k$, and therefore we add $|S_i|$ to count. If none of these two conditions is satisfied, then $D_k$ crosses $v$ and we visit all the children of $v$ in the next step. By repeating this step for all nodes in $V_i$, we have at our disposal all the nodes that we have to visit in the next step. It can be shown that during the simulation of the algorithm, the outcome of one of the oracle calls (computation of $|D(q, r) \cap S|)$ will be $r = r_i$, so the algorithm will return the value of $r_i$ before completing the simulation.

Since $\Psi$ has $O(\log n)$ levels, and we spend $O(n/\sqrt{s}) \log^c n$ time at each level, we can conclude that

**Theorem 3.6.** Let $S$ be a set of $n$ points in the plane, let $P$ be a convex polygon as defined above, and let $s$ be a parameter, with $n \leq s \leq n^x$. $S$ can be preprocessed in time $O(s \log n)$ into a data structure of size $O(s)$, so that the $k$th nearest neighbor for any point $q \in \partial P$ can be counted in $O(n/s) \log^c n$ time.

Going back to the problem of computing the $k$th nearest neighbor of $p_i$, we set $s = n^{1/3}$, preprocess $S$ as in Theorem 3.6, and query it with the pair $p_i, k_i$ for each $1 \leq i \leq n$.

**Theorem 3.7.** Let $S$ be a set of points in the plane in convex position, and let $k = (k_1, \ldots, k_n)$ be a vector of length $n$ as defined earlier. Then the $k$th nearest neighbor of $p_i$, for all $i \leq n$, can be computed in time $O(n^{4/3} \log^c n)$.

As for an arbitrary set of points in the plane, in view of Theorem 3.4, we can conclude

**Theorem 3.8.** Let $S$ be a set of points in the plane, and let $k = (k_1, \ldots, k_n)$ be a vector of length $n$ as defined earlier. Then the $k$th nearest neighbor of $p_i$, for all $i \leq n$, can be computed in time $O(n^{7/9} \log^c n)$.

4. CONCLUSION

In this paper, we presented efficient algorithms for the array- and row-selection problems. The running time of these algorithms can be improved if we have a faster procedure for decomposing a sequence into monotone sequences. As mentioned in Section 2, our algorithms work for generalized monotone matrices as well. As far as we know, this is the first subquadratic algorithm for the array-selection problem when $k$ is large.
We also presented and efficient algorithm for the all kth nearest neighbor problem for a set of points in convex position.

We conclude by mentioning some open problems.

1. Can the time complexity of the array-selection problem be improved to \(O(n^{4/3}\log^2 n)\)? Agarwal et al. [1] have given such an algorithm for selecting the kth smallest distance in a set of points in the plane?

2. Can the kth smallest distance in a set of planar points in convex position be computed in near-linear time? The best known algorithm to date is the same as for an arbitrary set of points (cf. [1]).

3. Can the all kth nearest neighbor problem for an arbitrary set of points in the plane be solved in time \(O(n^{4/3}\log^2 n)\)?

REFERENCES


