JOURNAL OF ALGORITHMS **20,** 581–601 (1996)
ARTICLE NO. 0028

Selection in Monotone Matrices and Computing kth Nearest Neighbors

Pankaj K. Agarwal*

Department of Computer Science, Duke University, P.O. Box 90129, Durham, North Carolina 27708-0129.

and

Sandeep Sen

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

Received March 8, 1994

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

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

We present an $O((m+n)\sqrt{n}\log n)$ time algorithm to select the kth smallest item from an $m \times n$ totally monotone matrix for any $k \le mn$. 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 *generalized row-selection problem* in monotone matrices. Given a set $S = \{p_1, \dots, p_n\}$ of n points in convex position and a vector $\mathbf{k} = \{k_1, \dots, k_n\}$, we also present an $O(n^{4/3}\log^c n)$ algorithm to compute the k_ith nearest neighbor of p_i for every $i \le n$; 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_ith nearest neighbor of p_i, for all $i \le n$, can be computed in time $O(n^{7/5}\log^c n)$, which also improves upon the previously best-known result. © 1996 Academic Press, Inc.

*Supported by an NYI award and National Science Foundation Grant CCR-93-01259.

1. INTRODUCTION

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

$$a_{i_1,j_1} < a_{i_1 j_2} \Rightarrow a_{i_2,j_1} < a_{i_2,j_2}. \tag{1}$$

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 left-most 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 $\mathscr{A}$, and a positive integer $k \le mn$, find the kth smallest element, x_k, of $\mathscr{A}$, i.e.,

$$\left|\left\{a_{i,j} \mid a_{i,j} < x_k, a_{i,j} \in \mathscr{A}\right\}\right| < k \text{ and } \left|\left\{a_{i,j} \mid a_{i,j} \le x_k, a_{i,j} \in \mathscr{A}\right\}\right| \ge k.$$

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

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

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.[1] Selecting the kth 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 kth 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 kth 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 $\mathscr{A}^T$ 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 $O(n\sqrt{m \log n} + m)$. Both of these algorithms rely on the fact that the value of k is the same for all rows of $\mathscr{A}$, so they do not extend to the following *generalized row-selection* problem: Given an $m \times n$ totally monotone matrix $\mathscr{A}$ and a vector $\mathbf{k} = \langle k_1, \dots, k_m \rangle$ of length m, where $0 < k_i \le n$, compute the k_ith smallest element of the ith row of $\mathscr{A}$, for all $1 \le i \le m$. Alon and Azar [10] proved that $O(n\sqrt{m} \log n\sqrt{\log m})$ 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 $\mathscr{A}^T$ 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* kth *nearest-neighbor* problem for a convex set: Given a set S of n points in the plane in convex position and a parameter $k \le n$, find the kth 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 kth nearest-neighbor problem for an arbitrary set of points in the plane directly. Using kth order Voronoi diagrams, the kth nearest neigh-

[1] In order to obtain an $o(mn)$ algorithm, we have to assume that $\mathscr{A}$ is not represented explicitly. We assume, that any particular entry $a_{i,j} \in \mathscr{A}$ can be computed in constant time. For instance, given a set $\{p_1, \dots, p_n\}$ of points, $a_{i,j}$ 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 $a_{i,j}$, for a given pair i, j, is $t(m, n)$, then we have to multiply the running time of the algorithm by a factor of $t(m, n)$.

bor of every point in an arbitrary planar point set can be computed in time $O(k^{3/2} n \log n)$. 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 $O(n^{3/2+\varepsilon})$ time algorithm for the all kth nearest neighbor problem, for any $\varepsilon > 0$ [2].

Here, we present an $O(n^{4/3} \log^c n)$-time algorithm, for some constant $c > 0$, for the all kth 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,[2] 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 ∂P (the boundary of P), the kth 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 $(n \le s \le n^2)$, so that a kth nearest neighbor query as described above, can be answered in time $O((n/\sqrt{s}) \log^c n)$ time, where $c \ge 1$ is some constant. By setting $s = n^{4/3}$ and querying the data structure with each point of S, the all kth 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 $O(n^{7/5} \log^c n)$.

2. SELECTION IN MONOTONE MATRICES

In this section we present algorithms for the array-selection and the generalized row selection problems. Let $\mathscr{A} = (a_{i,j})$, $1 \le i \le m$, $1 \le j \le n$, be an $m \times n$ totally monotone matrix, and let $k \le mn$ be a positive integer. The algorithm works in two steps. In the first step, it partitions $\mathscr{A}$ into a family $\mathscr{L} = \{L_1, \ldots, L_t\}$ of $t = O(m\sqrt{n})$ lists so that each list $L_i \in \mathscr{L}$ 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 i, the ith vertex of P can be accessed in $O(1)$ time.

The lists of $\mathscr{L}$ will be stored implicitly, as described below. After having computed the family of lists $\mathscr{L}$, we select the kth smallest element among the elements of lists of $\mathscr{L}$ in time $O(|\mathscr{L}| \log n) = O(m\sqrt{n} \log n)$, 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 $\mathscr{L}$. In addition to $\mathscr{L}$, we also compute a collection $\Pi = \{\pi_1, \dots, \pi_s\}$ of arrays, referred to as *index-arrays*, which will be used to represent $\mathscr{L}$ implicitly. Let $a_{r_i, k}$ be the lth item of L_i. For each L_i, we store the value of r_i and a pointer to an array $\pi_j \in \Pi$. The array π_j has the same size as L_i, and that the lth item of π_j is k if the lth item of L_i is $a_{r_i, k}$; we will denote π_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 $\mathscr{A}$ is a totally monotone matrix, then any submatrix of $\mathscr{A}$ 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) $a_{i,j} \geq a_{i,k} \Rightarrow a_{i',j} \geq a_{i',k}$ *for all $i' \leq i$, and*
(ii) $a_{i,j} \leq a_{i,k} \Rightarrow a_{i',j} \leq a_{i',k}$ *for all $i' \geq i$.*

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 $U = \langle u_1, u_2, \dots, u_m \rangle$ be a sequence of real numbers. A subsequence $U' = \langle u_{i_1}, u_{i_2}, \dots, u_{i_k} \rangle$ of U is called *monotone* if either $u_{i_1} \leq u_{i_2} \leq \cdots \leq u_{i_k}$ or $u_{i_1} \geq u_{i_2} \geq \cdots \geq u_{i_k}$.

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 $\mathscr{L}$. At each recursive step of the algorithm we have a $u \times v$ submatrix M of $\mathscr{A}$ (see Fig. 2); initially $M = \mathscr{A}$. M consists of a contiguous block of u rows of $\mathscr{A}$, say $\alpha + 1, \dots, \alpha + u$, and a subsequence $C(M)$ of columns of $\mathscr{A}$, i.e., $M = (a_{i,j})$, $\alpha + 1 \leq i \leq \alpha + u$, $j \in C(M)$. M is repre-

$$\begin{array}{l} M \\ \begin{array}{ccccccc} 59&47&20&37&7&11&2\\ 25&15&14&19&6&12&10\\ 40&33&24&34&22&28&24\\ 21&15&7&17&6&13&11\\ \hline \multicolumn{1}{|c}{32}&28&23&37&32&44&\multicolumn{1}{c|}{45}\\ \hline 88&85&76&92&90&100&110\\ 31&30&34&53&51&66&73\\ 6&9&13&32&32&52&62\\ 18&29&35&43&55&78&95 \end{array}\end{array} \Longrightarrow \begin{array}{l} L_1 = \langle 32,37,44,45\rangle \\ L_2 = \langle 88,92,100,110\rangle \\ L_3 = \langle 31,53,66,73\rangle \\ L_4 = \langle 6,32,52,62\rangle \\ L_5 = \langle 18,43,78,95\rangle \\ M_1 \begin{array}{ccccccc} 59&47&20&37&7&11&2\\ 25&15&14&19&6&12&10\\ 40&33&24&34&22&28&24\\ 21&15&7&17&6&13&11 \end{array} \\ \qquad\qquad M_2 \begin{array}{ccc} 28&23&32\\ 85&76&90\\ 30&34&51\\ 9&13&32\\ 29&35&55 \end{array} \end{array}$$

FIG. 2. Decomposing M into two M_1, M_2; $\sigma = \langle 32, 37, 44, 45\rangle$, $\pi_1 = \langle 1, 4, 6, 7\rangle$.

sented by specifying the values of α and u, and the subsequence $C(M)$. By Lemma 2.1, M is totally monotone matrix. Let $\mathscr{L}' = \{L_1, \dots, L_\xi\}$ and $\Pi' = \{\pi_1, \dots, \pi_\xi\}$ 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

$$\Pi' = \Pi' \cup \{\pi_{\zeta+1}\} \qquad \text{and} \qquad \mathscr{L}' = \mathscr{L}' \cup \left\{L_{\xi+1}, \dots, L_{\xi+u}\right\}.$$

where $\pi_{\zeta+1} = \langle z \rangle$ and, for $1 \le i \le u$, $L_{\xi+1} = \langle a_{(\alpha+i), z}\rangle$, $r_{\xi+1} = \alpha + i$, and $\varphi(L_{\xi+i}) = \pi_{\zeta+1}$.

Next, assume that $v > 1$ (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 wth row of M. Using Dijkstra's algorithm, we compute the longest monotone subsequence

$$\sigma = \langle a_{(\alpha+w), j_1}, a_{(\alpha+w), j_2}, \dots, a_{(\alpha+w), j_k} \rangle$$

of U. There are two cases to consider depending on whether σ is nonincreasing or nondecreasing.

Case A. Suppose σ is a monotonic nonincreasing subsequence, then for all x, $\alpha + 1 \le x \le \alpha + w$, the subsequence

$$\langle a_{x, j_1}, \dots, a_{x, j_k} \rangle$$

is also monotonic nonincreasing (cf. Lemma 2.2). We set

$$\Pi' = \Pi' \cup \{\pi_{\zeta+1}\} \qquad \text{and} \qquad \mathscr{L}' = \mathscr{L}' \cup \{L_{\xi+1}, \ldots, L_{\xi+w}\}, \tag{1}$$

where

$$\pi_{\zeta+1} = \langle j_k, j_{k-1}, \ldots, j_1 \rangle \tag{2}$$

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

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

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 $\mathscr{A}$, i.e., $\alpha + w + 1 \le i \le \alpha + u$, 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 σ, i.e., $\alpha + 1 \le i \le \alpha + w$, and $C(M_2) = C(M) - \langle j_1, \ldots, j_k \rangle$.

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

$$\Pi' = \Pi' \cup \{\pi_{\zeta+1}\} \qquad \text{and} \qquad \mathscr{L}' = \mathscr{L}' \cup \{L_{\xi+1}, \ldots, L_{\xi+u-w+1}\}. \tag{4}$$

where

$$\pi_{\zeta+1} = \langle j_1, j_2, \ldots, j_k \rangle, \tag{5}$$

$$L_{\xi+i+1} = \langle a_{(\alpha+w+i), j_1}, a_{(\alpha+w+i), j_2}, \ldots, a_{(\alpha+w+i), j_k} \rangle, \tag{6}$$

$r_{\xi+i+1} = \alpha + w + i$, and $\varphi(L_{\xi+i+1}) = \pi_{\zeta+1}$ for $0 \le i \le u - w$. $M_1 = (a_{i,j})$ is composed of the top $w - 1$ rows of M and all the columns of $\mathscr{A}$, i.e., $\alpha + 1 \le i \le \alpha + w - 1$ 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 σ, i.e., $\alpha + w \le i \le \alpha + u$ and $C(M_2) = C(M) - \langle j_1, \ldots, j_k \rangle$; see Fig. 2.

Next, let us analyze the size of $\mathscr{L}$ 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

$$\psi(u,v) \le \begin{cases} u & \text{if } v = 1. \\ \psi\left(\left\lfloor \dfrac{u}{2} \right\rfloor, v\right) + \psi\left(\left\lceil \dfrac{u}{2} \right\rceil, v - \lceil \sqrt{v}\, \rceil\right) + \left\lceil \dfrac{u}{2} \right\rceil & \text{if } v > 1. \end{cases}$$

We claim that the solution of this recurrence is

$$\psi(u,v) \le 2u\sqrt{v}\,. \tag{7}$$

Indeed, (7) is obviously true for $v = 1$. For $v > 1$, by induction hypothesis, we have

$$\begin{aligned}\psi(u,v) &\le \psi\left(\left\lfloor \frac{u}{2} \right\rfloor, v\right) + \psi\left(\left\lceil \frac{u}{2} \right\rceil, v - \sqrt{v}\right) + \left\lceil \frac{u}{2} \right\rceil \\ &\le 2\left\lfloor \frac{u}{2} \right\rfloor \sqrt{v} + 2\left\lceil \frac{u}{2} \right\rceil \sqrt{v - \sqrt{v}} + \left\lceil \frac{u}{2} \right\rceil \\ &\le u\left(\sqrt{v} + \sqrt{v - \sqrt{v}} + \frac{1}{2}\right) \\ &\le 2u\sqrt{v}\,,\end{aligned}$$

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 σ and $O(u + v)$ time in generating the lists plus the two submatrices M_1 and M_2, we have

$$T(u,v) \le T\left(\left\lfloor \frac{u}{2} \right\rfloor, v\right) + T\left(\left\lceil \frac{u}{2} \right\rceil, v - \sqrt{v}\right) + O(u + v \log v).$$

Following the same calculations as above, it can be shown that

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

This implies that $|\mathscr{L}| \le 2m\sqrt{n}\,$, and that the total time spent in computing $\mathscr{L}$ 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, $U_1, \dots, U_r$, each of length at most $\sqrt{v}\,$. Suppose U_j is a monotonic nonincreasing sequence. We update $\Pi', \mathscr{L}'$ 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 $\mathscr{L}', \Pi'$ 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, $1 \le j \le r$, we obtain the family $\mathscr{L}$. The recurrence for the maximum running time $T(u,v)$ now becomes

$$T(u,v) \le \sum_{i=1}^{r} T(\lfloor u/2 \rfloor, v_i) + O\big(u\sqrt{v} + v^{3/2}\big),$$

where $\max_{i \le r} v_i \le \sqrt{v}$ and $\sum_{i=1}^{r} v_i = v$. The solution of the above recurrence is easily seen to be

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

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 $\mathscr{L} = \{L_1, \dots, L_t\}$ of $O(m\sqrt{n})$ nondecreasing lists. $\mathscr{L}$ is stored implicitly so that for any pair (i, j), the jth element of L_i can be accessed in $O(1)$ time.*

2.2. *Selecting the kth Element*

Let $\mathscr{L} = \{L_1, \dots, L_t\}$ be a set of t lists, each sorted in nondecreasing order, and let us assume that, for any $i \le t$ and $j \le |L_i|$, the jth 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 kth smallest item in $\bigcup_{1 \le i \le t} L_i$.

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

In the ith set we first divide each list $\lambda \in \Lambda$ into two lists λ_1, λ_2: λ_1 consists of the first $m_i/2$ elements of Λ, and λ_2 consists of the next $m_i/2$ elements of Λ. Let $\overline{\Lambda}$ denote the set of lists after the divide step. Next, we discard some of the lists in $\overline{\Lambda}$ that are guaranteed not to contain the sth element of $\bigcup \overline{\Lambda}$.

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

(i) The list $L_j \in \mathscr{L}$ that contains λ, and

(ii) the index of the first item of λ in L_j.

When we split λ into λ_1, λ_2, this information for both λ_1, λ_2 can be computed in $O(1)$ time. The prune-step works in two stages. Let $\alpha = \lceil s/m_i \rceil + t$, and let $\lambda(i)$ denote the ith element of λ. We select the αth smallest element, x_α, from the set $A = \{\lambda(1) \mid \lambda \in \tilde{\Lambda}\}$; A can be obtained in $O(|\overline{\Lambda}|)$ time and x_α can be computed in another $O(|\overline{\Lambda}|)$ time. If $\alpha < |\overline{\Lambda}|$, we discard all $|\overline{\Lambda}| - \alpha$ lists of $\overline{\Lambda}$ whose minimum item is not smaller than x_α. Next, let $\beta = \lfloor s/m_i \rfloor - t$. If $\beta > 0$, we select the βth element, x_β, from the set $B = \{\lambda(m_i) \mid \lambda \in \overline{\Lambda}\}$, and discard all the lists whose maximum element is not larger than x_β. The second stage prunes at least β lists. Set $s = s - \beta m_i$.

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

Notice that after each phase, at most $2t$ lists are retained in $\overline{\Lambda}$, because at least $|\overline{\Lambda}| - \alpha + \beta \le |\overline{\Lambda}| - 2t$ lists are deleted by the prune step. The running time of the algorithm is thus $O(tp) = O(m\sqrt{n}\log n)$. The correctness of the algorithm follows from the fact that in the ith step, the sth element of $\bigcup \overline{\Lambda}$, 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 $\mathscr{A}$ be an $m \times n$ totally monotone matrix, and let $k \le mn$ be some positive integer. The kth smallest item of $\mathscr{A}$ 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 $\mathbf{k} = \langle k_1, k_2, \ldots, k_m \rangle$ be a vector of length m, where each $k_i \le n$ is a positive integer. The goal is to select the k_ith smallest item from the ith row of $\mathscr{A}$. We decompose $\mathscr{A}$ into a set $\mathscr{L}$ of nondecreasing lists as above. $\mathscr{L}$ is again represented implicitly using the set Π of index-arrays. Let

$$\mathscr{L}_i = \left\{ L_j \mid r_j = i \right\}$$

be the subset of lists belonging to the ith row of $\mathscr{A}$. We run the above selection algorithm for each $\mathscr{L}_i$ separately and select the k_ith item of $\mathscr{L}_i$. The running time for each $\mathscr{L}_i$ is $O(|\mathscr{L}_i|\log n)$, 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 $\mathscr{A}$ be an $m \times n$ totally monotone matrix and let $\mathbf{k}$ be a vector of length m as defined above. The k_ith smallest item of the ith row of $\mathscr{A}$, for all $1 \le i \le n$, 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 kth smallest geodesic distance between the vertices of P can be computed in time $O(n^{3/2}\log^2 n)$ time, and the kth 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 $k \le \binom{n}{2}$, the kth smallest geodesic distance between the vertices of P can be computed in time $O(n^{3/2}\log^2 n)$. For a given vector $\mathbf{k} = \langle k_1, \dots, k_n \rangle$, where each $k_i \le n$, the k_ith nearest neighbor of the ith vertex of P (over all $1 \le i \le n$) 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 kth NEAREST NEIGHBORS

Let S be a set of n points in the plane in convex position and let $\mathbf{k} = \langle k_1, k_2, \dots, k_n \rangle$ be a vector of length n, where each $k_i \le n$ is a positive integer. The goal is to compute the k_ith nearest neighbor of p_i for $i \le n$. 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 ∂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 kth nearest-neighbor queries for points on ∂P (i.e., for a query point $p \in \partial P$ and a positive integer $k \le n$, we wish to determine the kth 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 ℓ_{ij} denote the perpendicular bisector of p_i and p_j, and let

$$\mathscr{L} = \left\{ \ell_{ij} \mid 1 \le i < j \le n \right\}.$$

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 $d(q, p_{i_1}) \le d(q, p_{i_2}) \le \cdots \le d(q, p_{i_n})$, then

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

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 $\mathscr{A}(\mathscr{L})$, $\sigma(q) = \sigma(q')$. Abusing the notation slightly, let $\sigma(f)$ denote the distance-ordering with respect to any point in the face f of $\mathscr{A}(\mathscr{L})$.

LEMMA 3.1. *If f and f' are two adjacent faces in $\mathscr{A}(\mathscr{L})$ separated by the time ℓ_{kl}, then p_k, p_l are adjacent in $\sigma(f)$ and $\sigma(f')$. Moreover, if*

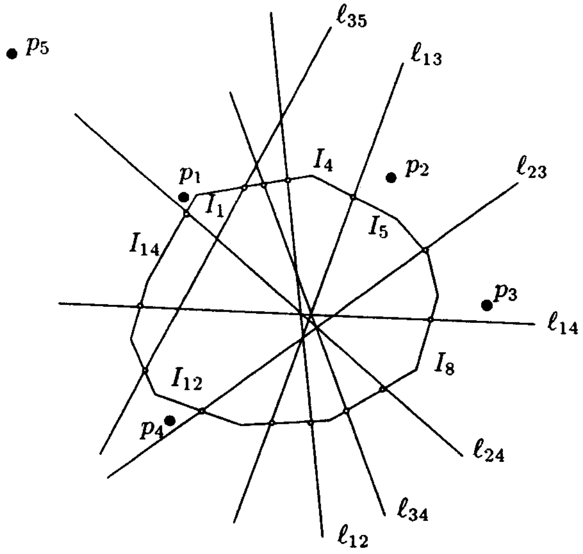
$$\sigma(f) = \langle p_{i_1}, \dots, p_{i_m}, p_k, p_l, p_{i_{m+3}}, \dots, p_{i_n} \rangle.$$

then

$$\sigma(f') = \langle p_{i_1}, \dots, p_{i_m}, p_l, p_k, p_{i_{m+3}}, \dots, p_{i_n} \rangle.$$

If we know the face f of $\mathscr{A}(\mathscr{L})$ 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 $\mathscr{A}(\mathscr{L})$, because there are $\Omega(n^4)$ faces in $\mathscr{A}(\mathscr{L})$. This is where we use the fact that the center of D lies on ∂P.

Partition ∂P into maximal connected intervals that do not intersect any line of $\mathscr{L}$, so that each interval lies within a face of $\mathscr{A}(\mathscr{L})$. Let $\mathscr{I} = \langle I_1, \dots, I_m \rangle$ be a sequence of the resulting intervals sorted along ∂P in clockwise direction; $m = O(n^2)$, for any line intersects ∂P in at most two points. Since each interval I_j lies within a single face of $\mathscr{A}(\mathscr{L})$, 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 ith leftmost node v stores the index i (which we refer to as the *rank* of v in T_j), and the ith 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 ∂P, $|S \cap D|$ can be computed in a straight-forward 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



$$\begin{aligned}\sigma(I_1) &= \langle 1,2,4,5,3\rangle\\ \sigma(I_2) &= \langle 1,2,4,3,5\rangle\\ \sigma(I_5) &= \langle 2,3,1,4,5\rangle\end{aligned}$$

FIG. 3. $S = \{p_1, p_2, p_3, p_4, p_5\}$, P, and $\mathscr{I} = \{I_1, \dots, I_{14}\}$.

persistent data structure to store all T_j'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 $\mathscr{D}$ denote the overall data structure; $\mathscr{D}$ is a dag that implicitly stores all T_j's in the sense that T_j is a subgraph of $\mathscr{D}$, 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 jth entry stores a pointer to a root of $\mathscr{D}$ that corresponds to the root of T_j. Each node $v \in \mathscr{D}$ 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.

$\mathscr{D}$ is constructed incrementally by processing the intervals of $\mathscr{I}$ one by one. Let $\mathscr{D}_j$ denote the data structures storing $T_1, \ldots, T_j$. $\mathscr{D}_1 = T_1$ is a minimum-height binary tree storing $\sigma(I_1)$. The ith leftmost node v_i of $\mathscr{D}_1$ stores the ith element of $\sigma(I_1)$, $rank(v_i) = i$, and $ver(v) = 0$. Roughly speaking, $\mathscr{D}_{j+1}$ can be obtained from $\mathscr{D}_j$ as follows: let ℓ_{kl} be the line separating I_j and I_{j+1}, and let u, v be the nodes corresponding to T_j that store p_k, p_l, respectively. We make new copies u', v' of u and v, and store p_k at v' and p_l at u' (i.e., swap p_k and p_l), 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 $\mathscr{D}_{j+1}$ from $\mathscr{D}_j$. The nodes u, v can be located in $O(\log n)$ time by searching T_j with p_k and p_l, respectively; see below for details of the search procedure. Since p_k and p_l 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 $val(x) = p_l$, $rank(x) = rank(u)$, $ver(x) = 0$. 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 $w = v$, then we set $val(w') = p_k$, otherwise we set $val(w') = val(w)$.

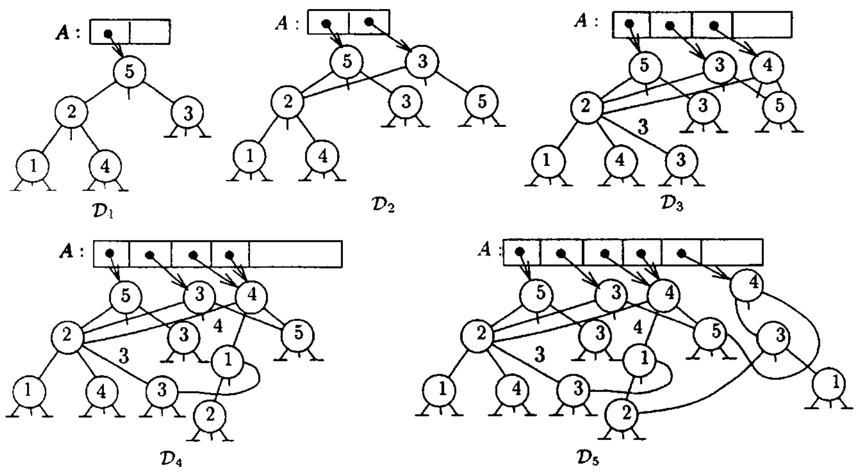


FIG. 4. $\mathscr{D}_1, \mathscr{D}_2, \mathscr{D}_3, \mathscr{D}_5$ for S and P shown in Figure 3; $\sigma(I_1) = \langle 1,2,4,5,3 \rangle$, $\sigma(I_2) = \langle 1,2,4,3,5 \rangle$, $\sigma(I_3) = \langle 1,2,3,4,5 \rangle$, $\sigma(I_4) = \langle 2,1,3,4,5 \rangle$, $\sigma(I_5) = \langle 2,3,1,4,5 \rangle$.

We also set $rank(w') = rank(w)$ and $ver(w') = 0$. The left and right pointers of w' are assigned in the same way as of x. If z is not the root of $\mathscr{D}$, then let y be the parent of z. If z is the left (resp. right) child of y, then we set $extra(y) = z$, $ver(y) = j + 1$, 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 $c \in \partial P$ and radius r. We compute $|D \cap S|$ as follows. We first find in $O(\log n)$ time the interval $I_j \in \mathscr{I}$ that contains c: $A[j]$ gives the pointer to the root z of T_j. We search in $\mathscr{D}$ 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 $rank(v)$. 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 $rank(v)$ (resp. $rank(v) - 1$). The left and right children of v in T_j can be computed in $O(1)$ time, as follows. If $0 < ver(v) \leq j$ 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 kth 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 ∂P can be counted in $O(\log n)$ time. The same data structure can also compute the kth 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" τ_v such that $S_v \subseteq \tau_v$; τ_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 $w_1, \dots, w_r$ are the children of v, then $\max_{1 \le i \le r} |S_{w_i}| \le 2|S_v|/r$ 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 τ_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 τ_v.

Let s be some fixed parameter for $n \le s \le n^2$. Suppose we want to construct a data structure of size s. Delete all those nodes of T whose parents are associated with less than $\lceil rs/n \rceil$ points. Let T' denote the resulting tree. Each leaf of T' contains at most $\lceil rs/n \rceil$ 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 Ψ 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 $v \in L$, $|S_v| \le \lceil rs/n \rceil$ and $\sum_{v \in L} |S_v| \le n$. 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 Ψ in a top-down fashion, starting from the root. We maintain a global variable *count*, which is initially set to 0. At each step, we visit a node v. If v is leaf of T', we compute $|D \cap S_v|$ using the algorithm of Theorem 3.2, and add this quantity to *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 *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 ∂D crosses at most $O(\sqrt{r}\log^{3/2} r)$ children of v, we obtain the following recurrence.

$$Q(n) \le \begin{cases} O(\log n) & \text{if } n_v \le s/n, \\ \displaystyle\sum_{i=1}^{O(\sqrt{r}\log^{3/2} r)} Q(n/r) + O(r) & \text{if } n_v > s/n. \end{cases}$$

The solution of the above recurrence is known to be

$$Q(n_v) = O\left(\frac{n^{1+\varepsilon}}{\sqrt{s}}\right),$$

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 $n \le s \le n^2$. 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 ∂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 $O(n^3)$ 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 $n \le s \le n^3$. 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 $k \le n$, determine its kth 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 Ψ level-by-level. Let V_i be the set of nodes that we visit in the ith step. If a node $v \in V_i$ belongs to the bottom structure, we need to determine the relation between r_k and $d(q, val(v))$ in order to determine whether the left or the right child of v has to be visited in the next step; let $\delta_v = d(q, val(v))$. If v is a node of the top structure, then we want to determine whether D_k crosses τ_v, D_k contains τ_v, or D_k is disjoint from τ_v. Let

$$\delta_v^- = \min_{p \in \tau_v} d(q, p) \qquad \text{and} \qquad \delta_v^+ = \max_{p \in \tau_v} d(q, p)$$

(see Fig. 5); D_k crosses τ_v if and only if $\delta_v^- < r_k < \delta_v^+$, and D_k contains τ_v if and only if $\delta_v^+ \le r_k$. The numbers δ_v^-, δ_v^+ can be computed in $O(1)$ time. Let

$$R_i = \{\delta_v^-, \delta_v^+ \mid v \in V_i \text{ and } v \text{ is a node of the top structure}\}$$
$$\cup \{\delta_v \mid v \in V_i \text{ and } v \text{ is a node of a bottom structure}\}.$$

We sort R_i and, by a binary search on the sorted list, we compute ρ_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 $|D(q, r) \cap S| \le k$, then $r < r_k$. If $|D(q, r) \cap S| = k$ and ∂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 $O((n/\sqrt{s}) \log^c n)$ 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 $v \in V_i$ of the bottom structure, if $\delta_v \le \rho_i$ (resp. $\delta_v > \rho_i$), i.e., $\delta_v < r_k$ (resp. $\delta_v > r_k$), then we visit the right (resp. left) child of v in the next step. For a node

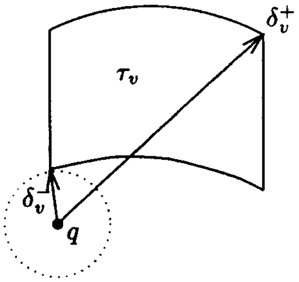


FIG. 5. $q, \delta_v^-, \delta_v^+$.

$v \in V_i$ of the top structure, if $\delta_v^- > \rho_i$ (i.e., $\delta_v^- > r_k$), then τ_v and D_k are disjoint, so we ignore v. If $\delta_v^+ \leq \rho_i$ (i.e., $\delta_v^+ < r_k$), then $\tau_v \subseteq D_k$, and therefore we add $|S_v|$ 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 $r = r_k$, so the algorithm will return the value of r_k before completing the simulation.

Since Ψ 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 $n \leq s \leq n^2$. S can be preprocesses in time $O(s \log n)$ into a data structure of size $O(s)$, so that the kth 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_ith nearest neighbor of p_i, we set $s = n^{4/3}$, 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 $\mathbf{k} = \langle k_1, \ldots, k_n \rangle$ be a vector of length n as defined earlier. Then the k_ith 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 $\mathbf{k} = \langle k_1, \ldots, k_n \rangle$ be a vector of length n as defined earlier. Then the k_ith nearest neighbor of p_i, for all $i \leq n$, can be computed in time $O(n^{7/5}\log^c n)$.*

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 $O(n^{4/3}\log^{O(1)} n)$? 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 $O(n^{4/3}\log^{c} n)$?

REFERENCES

1. P. K. Agarwal, B. Aronov, M. Sharir, and S. Suri, Selecting distances in the plane, *Algorithmica* **9** (1993), 495–514.
2. P. K. Agarwal and J. Matoušek, Ray shooting and parametric search, *SIAM J. Comput.* **22** (1993), 794–806.
3. P. K. Agarwal and J. Matoušek, Range searching with semi-algebraic sets, *Discrete Comput. Geom.* **11** (1994), 393–418.
4. A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber, Geometric applications of a matrix-searching algorithm, *Algorithmica* **2** (1987), 195–208.
5. A. Aggarwal and J. Park, Parallel searching in multidimensional monotone arrays, *J. Algorithms,* to appear.
6. A. Aggarwal and J. Park, Sequential searching in multidimensional monotone arrays, *J. Algorithms,* to appear.
7. A. Aggarwal and J. Park, Improved algorithms for economic lot size problems, *Oper. Res.* **41** (1993), 549–571.
8. A. Aggarwal, D. Kravets, J. Park, and S. Sen, Parallel searching in generalized Monge arrays with applications, *in* ‘‘Proc. 2nd ACM Symp. Parallel Algorithms and Architectures, 1990,’’ pp. 259–268.
9. A. Aggarwal and S. Suri, Computing the longest diagonal of a simple polygon, *Inform. Process. Lett.* **35** (1990), 13–18.
10. N. Alon and Y. Azar, Comparison-sorting and selecting in totally monotone matrices, *in* ‘‘Proceedings 3rd Annual ACM–SIAM Symposium on Discrete Algorithms, 1992,’’ pp. 403–408.
11. R. Bar Yehuda and S. Fogel, Good splitters with applications to ray shooting, *Algorithmica* **11** (1994), 133–145.
12. M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan, Time bounds for selection, *J. Comput. Systems Sci.* **7** (1973), 448–461.
13. P. Callahan and S. Kosaraju, Faster algorithms for some geometric graph problems in higher dimensions, *in* ‘‘Proceedings, 4th Annual ACM–SIAM Symposium on Discrete Algorithms, 1993,’’ pp. 291–300.
14. B. Chazelle, Cutting hyperplanes for divide-and-conquer, *Discrete Comput. Geom.* **10** (1993), 145–158.

15. B. Chazelle, R. Cole, F. Preparata, and C. Yap, New upper bounds for neighbor searching. *Inform and Control* **68** (1986), 105–124.
16. B. Chazelle, M. Sharir, and E. Welzl, Quasi-optimal upper bounds for simplex range searching and new zone theorems, *Algorithmica* **8** (1992), 407–429.
17. B. Chazelle and E. Welzl, Quasi-optimal range searching in space of finite VC-dimension. *Discrete Comput. Geom.* **4** (1989), 467–490.
18. M. Dickerson, R. Drysdale, and J. Sack, Simple algorithms for enumerating interpoint distances and finding k nearest neighbors, *Internat. J. Comput. Geom. Appl.* **2** (1992), 221–239.
19. E. Dijkstra, ‘‘A Discipline of Programming,’’ Prentice–Hall, Englewood Cliffs, NJ, 1976.
20. H. Edelsbrunner, ‘‘Algorithms in Combinatorial Geometry,’’ Springer-Verlag, Berlin, 1987.
21. P. Erdős and G. Szekeres, A combinatorial problem in geometry, *Compositio Math.* **2** (1935), 463–470.
22. D. Eppstein, Sequence comparison with mixed convex and concave sets, *J. Algorithms* **11** (1990), 85–101.
23. G. Frederickson and D. Johnson, Finding the kth shortest paths and p-centers by generating and searching good data structures, *J. Algorithms* **4** (1983), 61–80.
24. G. Frederickson and D. Johnson, Generalized selection and ranking: sorted matrices, *SIAM J. Comput.* **13** (1984), 14–30.
25. Z. Galil and J. Park, A linear-time algorithm for concave one-dimensional dynamic programming, *Inform. Process. Lett.* **33** (1990), 309–311.
26. L. Guibas and J. Hershberger, Optimal shortest path queries in a simple polygon, *J. Comput. Systems Sci.* **39** (1989), 126–152.
27. J. Hershberger and S. Suri, Matrix searching with the shortest path metric, *in* ‘‘Proceedings 25th Annual ACM Symposium on Theory of Computing, 1992,’’ pp. 485–494.
28. D. Johnson and T. Mitzoguchi, Selecting the kth element in $X + Y$ and $X_1 + X_2 + \cdots + X_m$, *SIAM J. Comput.* **7** (1978), 147–153.
29. M. Klawe and D. Kleitman, An almost linear time algorithm for generalized matrix searching, *SIAM J. Discrete Math.* **3** (1990), 81–97.
30. D. Kravets and J. Park, Selection and sorting in totally monotone arrays, *Math. Systems Theory* **24** (1991), 201–220.
31. Y. Mansour, J. Park, B. Schieber, and S. Sen, Improved selection in totally monotone arrays, *Internat. J. Comput. Geom. Appl.* **3** (1993), 115–132.
32. L. Larmore and B. Schieber, On line dynamic programming with applications to the prediction of RNA secondary structures, *J. Algorithms* **12** (1992), 490–515.
33. J. Matoušek and E. Welzl, Good splitters for counting points in triangles, *J. Algorithms* **13** (1992), 307–319.
34. N. Megiddo, Combinatorial optimization with rational objective functions, *Math. Oper. Res.* **4**, (1979), 414–424.
35. N. Megiddo, Applying parallel computation algorithms in the design of serial algorithms, *J. Assoc. Comput. Mach.* **30** (1983), 852–865.
36. N. Sarnak and R. E. Tarjan, Planar point location using persistent search trees, *Comm. ACM* **29** (1986), 609–679.
37. A. Schönhage, M. Paterson, and N. Pipinger, Finding the median, *J. Comput. Systems Sci.* **13** (1976), 184–199.
38. R. Wilber, The concave least-weight subsequence problem revisited, *J. Algorithms* **9** (1988), 418–425.
39. X. Wu, Optimal quantization by matrix searching, *J. Algorithms* **12** (1991), 663–673.
40. P. Vaidya, An $O(n \log n)$ algorithm for the all-nearest-neighbors problem, *Discrete Comput. Geom.* **4** (1989), 101–115.