Chapter 52

Selection and Sorting in Totally Monotone Arrays

Dina Kravets*+
James K. Park*+

Abstract

A two-dimensional array $A = \{a[i,j]\}$ is called *totally monotone* if for all $i_1 < i_2$ and $j_1 < j_2$, $a[i_1,j_1] < a[i_1,j_2]$ implies $a[i_2,j_1] < a[i_2,j_2]$. Totally monotone arrays were introduced by Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87], who showed that several problems in computational geometry and VLSI river routing could be reduced to the problem of finding a maximum entry in each row of a totally monotone array. In this paper, we consider several selection and sorting problems involving totally monotone arrays and give a number of applications of solutions for these problems. In particular, we obtain the following results for an $m \times n$ totally monotone array A:

1. The k largest entries in each row of A can be computed in $O(k(m+n))$ time. This allows us to determine the k farthest (or nearest) neighbors of each vertex of a convex n-gon in $O(kn)$ time.
2. Provided the transpose of A is also totally monotone, the k largest entries overall in A can be computed in $O(m + n + k\lg(st/k))$ time, where $s = \min\{k,m\}$ and $t = \min\{k,n\}$. This allows us to find the k farthest (or nearest) pairs of vertices of a convex n-gon in $O(n + k\lg(t^2/k))$ time, where $t = \min\{k,n\}$.
3. The rows of A can be sorted in $O(mn + n^2)$ time. This allows us to solve the following problem in $O(n^2(1+\lg\ell))$ time: given ℓ convex polygons with a total of n vertices, for all vertices v, sort the other vertices by distance from v.
4. Sorting all the entries of A requires $\Omega(mn\lg m)$ time.

*Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139

+Supported in part by the Air Force under Contract OSR-86-0076, the Army under Contract DAAL-03-86-K-0171, the Defense Advanced Research Projects Agency under Contracts N00014-89-J-1988 and N00014-87-K-0825, the Office of Naval Research under Contract N00014-86-K-0593, and an NSF Graduate Fellowship.

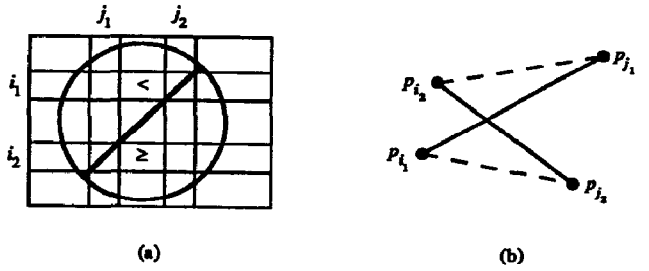

Figure 1.1: (a) In a totally monotone array, for no $i_1 < i_2$ and $j_1 < j_2$ is $a[i_1,j_1] < a[i_1,j_2]$ and $a[i_2,j_1] \geq a[i_2,j_2]$. (b) For any quadrilateral $p_{i_1}p_{i_2}p_{j_1}p_{j_2}$ where $i_1 < i_2 < j_1 < j_2$, $\mathrm{d}(p_{i_1},p_{j_1}) + \mathrm{d}(p_{i_2},p_{j_2}) > \mathrm{d}(p_{i_1},p_{j_2}) + \mathrm{d}(p_{i_2},p_{j_1})$.

1 Introduction

1.1 Motivation and Previous Work on Totally Monotone Arrays

A two-dimensional array $A = \{a[i,j]\}$ is called *monotone* if the maximum entry in its $(i+1)$-st row lies directly below or to the right of the maximum entry in its i-th row. (If a row has several maxima, then we consider only the leftmost one.) A is called *totally monotone* if every 2×2 subarray (*i.e.*, every 2×2 minor) is monotone. Equivalently, A is totally monotone if for all $i_1 < i_2$ and $j_1 < j_2$, $a[i_1,j_1] < a[i_1,j_2]$ implies $a[i_2,j_1] < a[i_2,j_2]$, as is suggested in Figure 1.1(a). Note that total monotonicity implies monotonicity.

Although the notion of a totally monotone array may seem rather odd at first glance, [AKM+87, AP89b, AP89a] have shown that such arrays arise naturally in connection with a wide variety of problems. As an example of this phenomenon (borrowed from [AKM+87]), consider a convex polygon P in the plane with vertices $p_1,\ldots,p_n$ in counterclockwise order. The distances separating pairs of vertices of P form a totally monotone array. Specifically, if we let $\mathrm{d}(p_i,p_j)$ denote the Euclidean distance from p_i to p_j, then the $n \times (2n-1)$

array $A = \{a[i,j]\}$, where

$$a[i,j] = \begin{cases} j-i & \text{if } 1 \le j < i, \\ \mathrm{d}(p_i, p_j) & \text{if } i \le j \le n, \\ \mathrm{d}(p_i, p_{j-n}) & \text{if } n < j < i+n, \\ -1 & \text{if } i+n \le j < 2n, \end{cases}$$

is totally monotone, as shown in [AKM+87] and [AK89]. Furthermore, the transpose A^T of A is also is totally monotone[1]. We refer to A as the *distance array* for P. The total monotonicity of A and A^T more or less follows from the *quadrangle inequality*: given any four vertices $p_{i_1}, p_{i_2}, p_{j_1}$, and p_{j_2} such that $1 \le i_1 < i_2 < j_1 < j_2 \le n$, the sum of the lengths of the diagonals of the quadrilateral formed by these vertices is *strictly greater* than the sum of the lengths of two opposite sides. In particular, $\mathrm{d}(p_{i_1}, p_{j_1}) + \mathrm{d}(p_{i_2}, p_{j_2}) > \mathrm{d}(p_{i_1}, p_{j_2}) + \mathrm{d}(p_{i_2}, p_{j_1})$, as suggested in Figure 1.1(b).

Totally monotone arrays were introduced by Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87], who showed that several problems in computational geometry and VLSI river routing could be reduced to the problem of finding a maximum entry in each row of a totally monotone array. (These entries will be referred to as *row maxima*.) Aggarwal et al. also gave a sequential algorithm that finds the leftmost maximum in each row of an $m \times n$ totally monotone array A in $\Theta(m)$ time when $m \ge n$ and in $\Theta(m(1+\lg(n/m)))$ time when $m < n$, provided each entry of A can be computed in constant time. Note that this algorithm does not explicitly create the entire array A (that would take $\Omega(mn)$ time); rather, it computes an entry only when that entry is needed. We will refer to this algorithm as the SMAWK algorithm, following the convention of [Wil88].

Returning to our example of the convex polygon in the plane, the SMAWK algorithm gives us a $\Theta(n)$ time algorithm for computing a farthest neighbor for each vertex of a convex n-gon. In [AK89], Aggarwal and Kravets extend this result, showing that *all* farthest neighbors for each vertex of a convex n-gon can be computed in $\Theta(n)$ time using the SMAWK algorithm. (In fact, their algorithm is easily modified to compute *all* maximum entries in each row of an $m \times n$ totally monotone array A in $O(n+m+s)$ time, where s is the output size, *i.e.*, the total number of row maxima.)

Another application of the totally monotone array abstraction is given by Wilber [Wil88], who solves the concave least-weight subsequence problem in linear time using the SMAWK algorithm. Aggarwal and Park [AP89b, AP89a] generalize the notion of totally monotone arrays to higher dimensions, develop new sequential and parallel algorithms for computing maxima in totally monotone arrays, and apply these algorithms, along with the SMAWK algorithm, to additional problems involving computational geometry, dynamic programming, VLSI river routing, and string editing. ([AP89b] presents sequential applications, while [AP89a] gives parallel applications.)

Note that SMAWK algorithm is easily adapted to computing a *minimum* entry in each row of a totally monotone array A. Conceptually, we need only negate the entries of A and reverse the ordering of its columns. The algorithms of [AP89b, AP89a, AK89] for two-dimensional totally monotone arrays may be modified in a similar fashion to solve the minimizing (rather than maximizing) variants of the respective problems.

1.2 Our Results

As indicated in the last subsection, previous work relating to totally monotone arrays was limited to maximization (or minimization) problems. In this paper, we consider two more comparison problems in the context of totally monotone arrays: selection and sorting. Given n values $a_1, \ldots, a_n$ and an integer k between 1 and n, the selection problem is that of finding a k-th largest value, *i.e.*, an a_i such that $|\{a_j : a_j \ge a_i\}| \ge k$ and $|\{a_j : a_j \le a_i\}| \ge n-k+1$. Given n values $a_1, \ldots, a_n$, the sorting problem is that of finding a permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that $a_{\sigma(1)} \ge a_{\sigma(2)} \ge \cdots \ge a_{\sigma(n)}$.

For arbitrary values $a_1, \ldots, a_n$, the selection and sorting problems are well understood: the general selection problem can be solved in $\Theta(n)$ time [BFP+72] and the general sorting problem in $\Theta(n \lg n)$ time (see [Knu73], for example). Using the special structure of totally monotone arrays, we obtain significantly better results for certain selection and sorting problems involving such arrays than are possible with the classical selection and sorting algorithms. We then apply these results to a number of problems involving convex polygons in the plane.

The remainder of this paper is organized as follows.

In Section 2, we consider the problem of computing a k-th largest entry in each row of a totally monotone array A. We call this the *row selection* problem for A. For an $m \times n$ array A, we show that the row selection problem can be solved in $O(k(m+n))$ time. For small values of k, this represents a significant improvement over the naive $O(mn)$ time algorithm obtained by applying the linear time selection algorithm of [BFP+72] m times[2]. We also show how our row selection algorithm can be used to compute k farthest or k nearest neighbors for each vertex of a planar convex n-gon in $O(kn)$ time. Previous

[1] As a technical detail, we need the negative entries of A (*i.e.*, those that do not correspond to distances) to make A and A^T totally monotone, though they render A's definition somewhat cumbersome.

[2] Recently, Mansour, Schieber, and Sen [MSS89] have obtained an $O(m^\alpha n)$ time algorithm for the row selection problem, where α is a constant slightly less than 1. However, for small values of k, our algorithm remains the best known.

results for this problem include an $O(n^{9/5}\lg n)$ time algorithm suggested by Chazelle [Cha87], an $O(kn^{3/2}\lg n)$ time algorithm based on the k-th order Voronoi diagram algorithm of [Ede86], and an $O(k^2n + n\lg n)$ time algorithm based on the k-th order Voronoi diagram algorithm of [Lee82, AGSS87]. All three of these results apply to arbitrary sets of points in the plane; thus, they are more general than our algorithm for computing k farthest neighbors. However, for the vertices of a convex polygon, our algorithm is superior when $k = O(n^{9/5})$.

In Section 3, we consider the problem of computing a k-th largest entry overall in a totally monotone array A. We call this the *array selection* problem for A. For an $m \times n$ array A, we show that the array selection problem can be solved in $O(m + n + k\lg(st/k))$ time, where $s = \min\{k, m\}$ and $t = \min\{k, n\}$. For small values of k, this again represents a significant improvement over the naive $O(mn)$ time algorithm obtained by applying the linear time selection algorithm of [BFP+72]. We also show how our array selection algorithm can be used to compute k farthest or k nearest pairs of vertices of a planar convex n-gon in $O(n + k\lg(t^2/k))$ time, where $t = \min\{k, n\}$. The best previous result for this problem is the $O(n^{9/5}\lg n)$ time algorithm of Chazelle [Cha87]. His algorithm is again more general than ours, since it applies to arbitrary sets of points in the plane. However, for the vertices of a convex polygon, our algorithm is superior when $k = O(n^{9/5})$.

In Section 4, we consider the problem of sorting the rows of a totally monotone array A. We call this the *row sorting* problem for A. For an $m \times n$ array A, we show that the row sorting problem can be solved in $O(mn+n^2)$ time. For $n = O(m\lg m)$, this represents an improvement over the naive $O(mn\lg n)$ time algorithm obtained by applying a general sorting algorithm to each row of A. As an application of our row sorting algorithm, we show that, given a convex n-gon P in the plane, for all vertices v of P, we can sort the other vertices by distance from v in $O(n^2)$ time. We then generalize this algorithm to ℓ polygons with a total of n vertices, showing that for each vertex v, we can sort the other vertices by distance from v in $O(n^2(1 + \lg\ell))$ time. The $\ell = 1$ result allows us to find all triples of vertices from a convex polygon forming isosceles triangles in $O(n^2)$ time, which settles an open question raised by Guibas [Gui88].

In Section 5, we consider the problem of sorting all the entries of a totally monotone array A. We call this the *array sorting* problem for A. For an $m \times n$ array A, we show that the array sorting problem requires $\Omega(mn\lg m)$ time. Thus, for $m = \Theta(n)$, the total monotonicity of A does not make sorting the entries of A any easier than sorting mn arbitrary values. Note that this lower bound implies there is no straightforward way of using totally monotone arrays to sort in $O(n^2)$ time the $\binom{n}{2}$ Euclidean distances separating n points in the plane, even if the points are the vertices of a convex polygon in counterclockwise order. This problem remains open. (If the L_1 metric is used in place of the L_2 metric, then an $O(n^2)$ time solution for the problem is known [Fre76].)

Finally, in Section 6, we present some open problems.

In the following discussion, we assume all the entries in our totally monotone arrays are distinct. This is merely to simplify our presentation; all the algorithms and analyses presented in this paper are easily modified to handle equalities.

2 Row Selection

2.1 A Row Selection Algorithm

In this subsection, we describe an algorithm that, given an $m \times n$ totally monotone array $A = \{a[i,j]\}$ and an integer k between 1 and n, computes the k largest entries in each row of A in $O(k(m + n))$ time. The algorithm combines two previous results with an important property of totally monotone arrays to achieve the specified time bounds. The first of these previous results is the SMAWK algorithm, described in the introduction. The second is the selection algorithm of Frederickson and Johnson [FJ82], that computes, as a special case, the k largest elements overall in $O(k)$ sorted lists in $O(k)$ time. The property of totally monotone arrays linking these two algorithms is given in the following lemma.

Lemma 2.1 *Let $B = \{b[i,j]\}$ be an $m \times n$ totally monotone array, where $m \geq n$. If each column of B contains at least one row maximum, then each row of B is bitonic. Specifically, for $1 \leq i \leq m$,*

$$b[i,1] < \cdots < b[i,c(i)-1] < b[i,c(i)]$$

and

$$b[i,c(i)] > b[i,c(i)+1] > \cdots > b[i,n],$$

where $c(i)$ denotes the column containing the maximum entry in row i.

Proof Suppose each column of B contains at least one row maximum, but B is not bitonic. Since B is not bitonic, there exist indices i, j_1, and j_2, such that $1 \leq i \leq m$, $1 \leq j_1 < j_2 \leq n$, and either

1. $j_1 < j_2 \leq c(i)$ and $b[i,j_1] > b[i,j_2]$, or
2. $c(i) \leq j_1 < j_2$ and $b[i,j_1] < b[i,j_2]$.

We consider only the first possibility; the proof for the second possibility is analogous. Since each column of B contains at least one row maximum, there exists an i' such that $c(i') = j_2$. Furthermore, $c(i') \neq c(i)$, since by assumption, $b[i,j_1] > b[i,c(i')]$, but by definition, $b[i,c(i)]$ is the maximum entry in row i. We must also

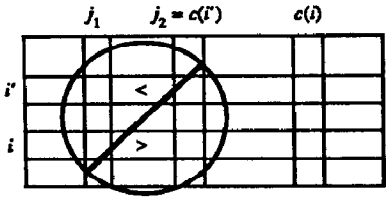

Figure 2.1: If the maximum entry in row i' lies in column j_2, then by the total monotonicity of B, we cannot have $b[i, j_1] > b[i, j_2]$.

have $i' < i$, since total monotonicity implies monotonicity. Now consider the 2×2 minor of B corresponding to rows i' and i and columns j_1 and j_2. (This subarray is depicted in Figure 2.1.) By definition, $b[i', c(i')]$ is the maximum entry in row i'. Thus, $b[i', j_1] < b[i', c(i')]$. By assumption, $b[i, j_1] > b[i, c(i')]$. This contradicts the total monotonicity of B. ■

We now sketch our algorithm for computing the k largest entries in each row of A. The algorithm has two parts. First, we decompose A into a series of m-row subarrays $B_1, \ldots, B_k$. The first subarray B_1 consists of those columns of A that contain row maxima of A. If we let A_1 be the m-row subarray of A consisting of those columns of A not in B_1, then B_2 consists of those columns of A_1 that contain row maxima of A_1. In general, if we let $A_{\ell-1}$ be the m-row subarray of A consisting of those columns of A not in any of $B_1, \ldots, B_{\ell-1}$, then B_ℓ consists of those columns of $A_{\ell-1}$ that contain row maxima of $A_{\ell-1}$. Using the SMAWK algorithm, we can compute $B_1, \ldots, B_k$ (or, more precisely, the columns forming these arrays) in $O(k(m+n))$ total time.

Since the row maxima of B_ℓ, $1 \leq \ell \leq k$, are the row maxima of $A_{\ell-1}$ (where, by convention, $A_0 = A$), each column of B_ℓ must contain at least one row maximum; thus, by Lemma 2.1, the rows of B_ℓ are bitonic. Furthermore, if an entry is among the ℓ largest entries in some row of A, $1 \leq \ell \leq k$, then the entry must be contained in one of $B_1, \ldots, B_\ell$. Thus, to compute the k largest entries in row i of A, we merely need to compute the k largest elements in the $2k$ sorted lists associated with row i. (Each B_ℓ contributes two sorted lists, the first consisting of those entries in the i-th row of B_ℓ to the right of the row's maximum and the second consisting of those entries to the maximum's left.) This can be accomplished in $O(k)$ time using the selection algorithm given by Frederickson and Johnson in [FJ82]. Since A contains m rows, the total time for this second part of the algorithm is $O(km)$, which gives the entire row selection algorithm a running time of $O(k(m+n))$.

Note that our algorithm does not output the k largest entries in a row of A in sorted order, as the algorithm of [FJ82] does not provide its output in sorted order. Also note that the size of our algorithm's output, km, is not necessarily a lower bound on the time required for the row selection problem; there may be a more concise representation for the output, given the highly structured nature of totally monotone arrays. Finally, note that our row selection algorithm can also be used to find the k smallest entries in each row of a totally monotone array; as suggested in the introduction, we merely negate each entry of the array and reverse the ordering of its columns.

2.2 Applications of Row Selection

Using the row selection algorithm of the previous subsection, we can solve two selection problems involving convex polygons in the plane. Given a set $S = \{p_1, \ldots, p_n\}$ of n points in the plane and an integer k between 1 and n, the *k farthest neighbors* problem for S is that of computing k farthest neighbors for each point p_i. More precisely, for all i between 1 and n, we must find a subset $S_i \subset S$ such that $|S_i| = k$ and for all $q \in S_i$ and $q' \in S - S_i$, $\mathrm{d}(p_i, q) \geq \mathrm{d}(p_i, q')$. The *$k$ nearest neighbors* problem for S is defined analogously[3]. If the points $p_1, \ldots, p_n$ are the vertices of a convex n-gon in counterclockwise order, then using our algorithm for computing the k largest entries in each row of a totally monotone array, we can solve both the k farthest neighbors problem and the k nearest neighbors problem for $p_1, \ldots, p_n$ in $O(kn)$ time.

To reduce the k farthest neighbors problem for $p_1, \ldots, p_n$ to a row selection problem, we use the $n \times (2n-1)$ totally monotone distance array A defined in Section 1. As the n largest entries in row i of A are the n distances $\mathrm{d}(p_i, p_1), \mathrm{d}(p_i, p_2), \ldots, \mathrm{d}(p_i, p_n)$, we can use our row selection algorithm to solve the k farthest neighbors problem for $p_1, \ldots, p_n$ in $O(kn)$ time.

To solve the k nearest neighbors problem for $p_1, \ldots, p_n$, we would like to reuse the array A defined above; however, to compute the k nearest neighbors of p_i, we need the $n+k$ smallest entries in row i, since the n smallest entries in this row are negative integers that do not correspond to distances. For $1 \leq k \leq \lfloor n/2 \rfloor$, our upper bound on the time to compute the $n+k$ smallest entries in A is $O(n^2)$. To obtain an $O(kn)$ time algorithm for the k nearest neighbors problem, we need a slightly more complicated reduction. (Note that we cannot circumvent this difficulty by replacing the negative integers in A with large positive integers, as this destroys the total monotonicity of A.)

In [LP78], Lee and Preparata consider the *nearest neighbor* problem (the $k = 1$ special case of the k nearest neighbors problem) for the vertices of a convex n-gon. In obtaining an $O(n)$ time solution to this problem, they introduce an interesting property of certain convex polygons which they call the *semicircle property*.

[3] As the k nearest neighbors problem for S is equivalent to the $n-k$ farthest neighbors problem for S, we restrict our attention to values of k between 1 and $\lfloor n/2 \rfloor$ for both problems.

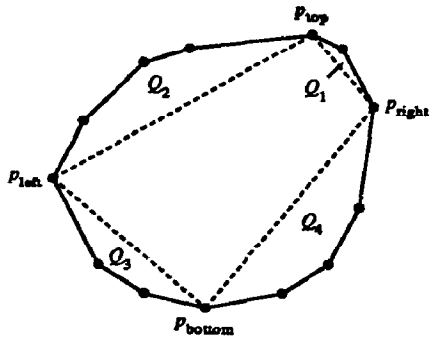

Figure 2.2: Q_1, Q_2, Q_3, and Q_4 have the semi-circle property.

A convex polygon P with vertices $p_1, \ldots, p_n$ in counterclockwise order is said to possess the semicircle property if $p_2, \ldots, p_{n-1}$ lie inside the circle with diameter $\overline{p_1 p_n}$.

Lemma 2.2 ([LP78]) *If P satisfies the semicircle property, then for all i between 1 and n, the sequence of distances $d(p_i, p_1), d(p_i, p_2), \ldots, d(p_i, p_n)$ is bitonic. Specifically,*

$$d(p_i, p_1) > \cdots > d(p_i, p_{i-1}) > d(p_i, p_i)$$

and

$$d(p_i, p_i) < d(p_i, p_{i+1}) < \cdots < d(p_i, p_n).$$

Lee and Preparata also showed how to decompose an arbitrary convex n-gon into four convex polygons possessing the semicircle property. We use a slightly simpler decomposition, due to Yang and Lee [YL79]:

Lemma 2.3 ([YL79]) *Let p_{left} and p_{right} be the vertices of P with minimum and maximum x-coordinates, respectively, and let p_{bottom} and p_{top} be the vertices of P with minimum and maximum y-coordinates, respectively. Let Q_1 be the polygon formed by vertices p_{right} through p_{top} (i.e., p_{right}, p_{top}, and those vertices between p_{right} and p_{top} in the counterclockwise ordering of P's vertices). Similarly, let Q_2, Q_3, and Q_4 be the polygons formed by vertices p_{top} through p_{left}, p_{left} through p_{bottom}, and p_{bottom} through p_{right}, respectively, as shown in Figure 2.2. Q_1, Q_2, Q_3, and Q_4 possess the semi-circle property.*

Using this decomposition of P, we can compute the k nearest neighbors of each vertex of P. We restrict our attention to the vertices of Q_1, showing that their k nearest neighbors in P can be computed in $O(kn)$ time — the computation of the k nearest neighbors of the vertices of Q_2, Q_3, and Q_4 is analogous. For each v in Q_1, the k nearest neighbors of v in Q_1 can be computed in $O(k)$ time, since by the semi-circle property, these k nearest neighbors must be within k of v in the original ordering of P's vertices. We can also compute for each v in Q_1 its k nearest neighbors in Q_2. To do this, we consider the $|Q_1| \times (|Q_2| - 1)$ array $A = \{a[i,j]\}$ where $a[i,j]$ is the distance from the i-th vertex of Q_1 to the $(j-1)$-st vertex of Q_2. (We ignore the first vertex of Q_2 since it is also the last vertex of Q_1.) It is readily verified that A is totally monotone; moreover, the k smallest entries in row i of A correspond to the k nearest neighbors in Q_2 of the i-th vertex of Q_1. Thus, using our row selection algorithm, we can find the k nearest neighbors in Q_2 of all the vertices in Q_1 in $O(kn)$ total time. In a similar manner, we can compute for each v in Q_1 its k nearest neighbors in Q_3 and its k nearest neighbors in Q_4. We now have $4k$ neighbors for each v in Q_1; using the linear time selection algorithm of [BFP+72], we can select the k nearest of these neighbors in $O(k)$ additional time. This gives the k nearest neighbors in P of each v in Q_1 in $O(kn)$ total time.

3 Array Selection

3.1 An Array Selection Algorithm

In this subsection, we describe an algorithm that, given an $m \times n$ array $A = \{a[i,j]\}$, such that both A and its transpose A^T are totally monotone, and an integer k between 1 and mn, computes the k largest entries overall in A in $O(m+n+k\lg(st/k))$ time, where $s = \min\{m,k\}$ and $t = \min\{n,k\}$. We first present an algorithm for those values of k that are greater than or equal to both m and n and then show how to modify this algorithm to handle smaller values of k.

To compute the k largest entries of A, $\max\{m,n\} \leq k \leq mn$, we begin by checking the relative magnitudes of k and mn. If $k \geq mn/2$ (the "easy" case), we use the linear time selection algorithm of [BFP+72] to compute the k-th largest entry of A in $O(k)$ time. If, on the other hand, $k < mn/2$, we consider two subcases.

If $m \geq n$, we use the row selection algorithm of Section 2 to compute the $2k/m$ largest entries in each row of A in $O((2k/m)(n+m)) = O(k)$ time. Let b_i denote the $(2k/m)$-th largest entry in row i of A. Using the linear time selection algorithm, we can compute the median b^* of $b_1, \ldots, b_m$ in $O(m)$ time. Let B denote the $m/2 \times n$ subarray of A consisting of those rows i such that $b_i \geq b^*$, and let L denote the list of $(2k/m)(m/2) = k$ entries formed from the $2k/m$ largest entries of each row of A not in B. Now if row i of A is not in B, *i.e.*, $b_i < b^*$, then the $n - (2k/m)$ smallest entries in row i are all smaller than b^*, which means they are all smaller than the $2k/m$ largest entries in each row of B. Since B has $m/2$ rows, this means that the $n - (2k/m)$ smallest entries in row i are all smaller than at least $(m/2)(2k/m) = k$ other entries, *i.e.*, these entries need not be considered as can-

didates for the k largest entries overall of A. Thus, if we recursively compute the k largest entries in B and then use the linear time selection algorithm to compute in $O(k)$ time the k largest of these entries and the k entries of L, we obtain the k largest entries in A.

If $m < n$, we apply the procedure described in the last paragraph to A^T rather than A. This requires $O(k)$ time plus the time needed to recursively compute the k largest entries in an $m \times n/2$ subarray of A.

Letting $T(k,m,n)$ denote our algorithm's running time in computing the k largest entries in an $m \times n$ array A, where $\max\{m,n\} \leq k \leq mn$ and both A and A^T are totally monotone, we have

$$T(k,m,n) = \begin{cases} O(k) & \text{if } k \geq mn/2, \\ T(k,m/2,n) & \\ \quad +O(k) & \text{if } k < mn/2 \\ & \text{and } m \geq n, \\ T(k,m,n/2) & \\ \quad +O(k) & \text{if } k < mn/2 \\ & \text{and } m < n. \end{cases}$$

The solution to this recurrence is

$$T(k,m,n) = O(k \lg(mn/k)).$$

Now suppose $k < m$. We can eliminate all but k of A's rows from consideration as follows. In $O(n+m)$ time, we can compute the row maxima of A. Then, using the linear time selection algorithm, we can select the k largest of these maxima in an additional $O(m)$ time. Now consider the $m-k$ rows of A corresponding to the $m-k$ smallest row maxima. The entries in these rows are all smaller than the k largest row maxima, which means they are not among the k largest entries of A. Thus, we can eliminate these $m-k$ rows from consideration. Similarly, if $k < n$, we can eliminate all but k of A's columns in $O(n+m)$ time.

Once the number of rows in A has been reduced to k or less and the number of columns in A has been reduced to k or less, we can apply our $O(k \lg(mn/k))$ time selection algorithm for arrays with $m \leq k$ rows and $n \leq k$ columns. This gives an algorithm for computing the k largest entries in A that works for all values of k between 1 and mn and runs in $O(m+n+k \lg(st/k))$ time, where $s = \min\{m,k\}$ and $t = \min\{n,k\}$.

Note that the only lower bound we have on the time required for the array selection problem is $\Omega(n)$. Also note that our array selection algorithm can also be used to compute the k smallest entries overall in a totally monotone array, just as our row selection algorithm can be used to compute the k smallest entries in each row of a totally monotone array.

3.2 Applications of Array Selection

Using the array selection algorithm of the previous subsection, we can solve two more selection problems involving convex polygons in the plane. Given a set $S = \{p_1, \ldots, p_n\}$ of n points in the plane and an integer k between 1 and $\binom{n}{2}$, the *k farthest pairs* problem for S is that of computing k largest values of $d(p_i,p_j)$ over all unordered pairs (p_i,p_j) of points. The *k nearest pairs* problem for S is defined analogously. If the points $p_1, \ldots, p_n$ are the vertices of a convex n-gon in counterclockwise order, then using our algorithm for computing the k largest entries overall in a totally monotone array, we can solve both the k farthest pairs problem and the k nearest pairs problem for $p_1, \ldots, p_n$ in $O(n + k \lg(t^2/k))$ time, where $t = \min\{n,k\}$.

To reduce the k farthest pairs problem and the k nearest pairs problem for $p_1, \ldots, p_n$ to row selection problems, we use constructions similar to those used in the last section. (Here we make use of the total monotonicity of the transpose of the distance array associated with a convex polygon.) For the sake of brevity, we omit the details of these reductions.

4 Row Sorting

4.1 A Row Sorting Algorithm

In this subsection, we sketch an algorithm for sorting the rows of an $m \times n$ totally monotone array $A = \{a[i,j]\}$ in $O(mn+n^2)$ time.

For $1 \leq i \leq m$ and $1 \leq r \leq n$, let $c_r[i]$ denote the column of the entry in row i of A with rank r in row i. Furthermore, for $1 \leq r \leq n$, let $c_r[0] = n-r+1$. Our algorithm consists of m phases, where in the i-th phase, we sort row i of A by computing $c_1[i], c_2[i], \ldots, c_n[i]$ using $c_1[i-1], c_2[i-1], \ldots, c_n[i-1]$. Specifically, we use an insertion sort (such as the one described in [Knu73]) to sort row i, inserting first $a[i, c_1[i-1]]$, then $a[i, c_2[i-1]]$, then $a[i, c_3[i-1]]$, and so on through $a[i, c_n[i-1]]$.

As noted in [Knu73], the insertion sort algorithm sorts n values in $O(n+I)$ time, where I is the number of inversions separating the insertion order and the final sorted order for the values. An inversion is a pair of values (a,b) such that a is inserted before b but $a < b$ (or vice-versa). It is not hard to verify that the total number of inversions encountered in stepping through the rows of an $m \times n$ totally monotone array $A = \{a[i,j]\}$ is $O(n^2)$, since for each pair of columns in A, corresponding to indices j_1 and j_2, $1 \leq j_1 < j_2 \leq n$, there exists at most one index i, $1 \leq i < m$, such that $a[i,j_1] > a[i,j_2]$ and $a[i+1,j_1] < a[i+1,j_2]$. Thus, our algorithm's running time is $O(mn+n^2)$.

Note that the size of our algorithm's output, mn, is not necessarily a lower bound on the time required for the row sorting problem; there may be a more concise representation for the output, given the highly structured nature of totally monotone arrays.

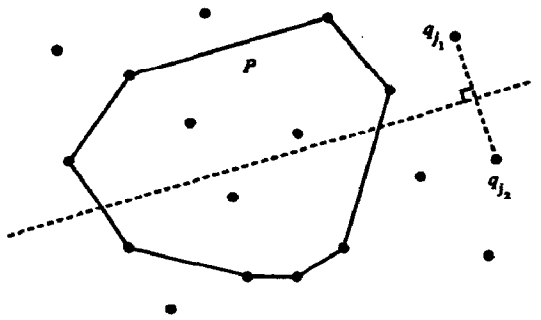

Figure 4.1: The perpendicular bisector of any pair of points q_{j_1} and q_{j_2} can intersect P at most twice.

4.2 Applications of Row Sorting

As an application of our row sorting algorithm, we consider the following *point sorting* problem: given a convex n-gon P with vertices $p_1, \ldots, p_n$ in counterclockwise order, for each p_i, sort the other vertices of P by distance from p_i. Recall the $n \times (2n-1)$ totally monotone distance array $A = \{a[i,j]\}$ we defined in Section 1. The i-th row of A contains the distances $\mathrm{d}(p_i, p_1), \ldots, \mathrm{d}(p_i, p_n)$, along with n negative entries; thus, sorting the rows of A using our row sorting algorithm gives an $O(n^2)$ time solution to the point sorting problem for P.

Now consider the following more general point sorting problem: given a convex m-gon P with vertices $p_1, \ldots, p_m$ in counterclockwise order and a set Q of n arbitrary points $q_1, \ldots, q_n$, for each p_i, sort the vertices of Q by distance from p_i. We call this the problem of *sorting Q with respect to P*. We cannot represent the distances separating the vertices of P and the points of Q as a totally monotone array. However, the correctness of our row sorting algorithm does not depend on total monotonicity; total monotonicity is merely used to bound of the number of inversions. Thus, even though the $m \times n$ array $B = \{b[i,j]\}$, where $b[i,j] = \mathrm{d}(p_i, q_j)$, is not totally monotone, we can still apply our row sorting algorithm

In the context of the array B, an inversion corresponds to indices i, j_1, and j_2 such that $b[i,j_1] < b[i,j_2]$ but $b[i+1,j_1] > b[i+1,j_2]$. To bound the number of times this can occur for any particular pair of indices j_1 and j_2, note that the perpendicular bisector of q_{j_1} and q_{j_2} can intersect P at most twice, as shown in Figure 4.1. Since there is an inversion for j_1 and j_2 between rows i and $i+1$ if and only if the perpendicular bisector of q_{j_1} and q_{j_2} intersects the edge of P connecting p_i and p_{i+1}, there are at most two inversions associated with the pair (j_1, j_2). Since there are $\binom{n}{2}$ pairs of points in Q, the total number of inversions is no more than $2\binom{m}{2} = O(n^2)$. Thus, we can sort the rows of B (*i.e.*, sort P with respect to Q) in $O(mn + n^2)$ time.

Taking the point sorting problem one step further, suppose we are given a set P of n arbitrary points $p_1, \ldots, p_n$ and that we want to sort P with respect to itself. In other words, for each point p_i, we want to sort the other points by distance form p_i. To solve this problem, we need to partition the points of P into subsets forming convex polygons $P_1, \ldots, P_\ell$. (This could be accomplished by computing the convex layers of P, which requires only $O(n \lg n)$ time [Cha85].) Assuming this partition is given, we can sort P with respect to itself in $O(n^2(1 + \lg \ell))$ time as follows. Let $P'_1, \ldots, P'_\ell$ be a second partition of the points of P into ℓ arbitrary subsets, each of size n/ℓ. Then for $1 \le i \le \ell$ and $1 \le j \le \ell$, we sort P'_i with respect to P_j in $O(n_j n/\ell + n^2/\ell^2)$ time, where n_j is the size of P_j. The total time required is

$$\sum_{i=1}^{\ell} \sum_{j=1}^{\ell} O\left(n_j \frac{n}{\ell} + \frac{n^2}{\ell^2}\right) = \sum_{i=1}^{\ell} O\left(\frac{n^2}{\ell}\right) = O(n^2).$$

For each point p_i of P, we now have ℓ sorted lists, corresponding to the points of $P'_1, \ldots, P'_\ell$, respectively. As we can merge these lists in $O(n^2 \lg \ell)$ total time, we obtain an $O(n^2(1 + \lg \ell))$ time algorithm for sorting the points of P.

As a final application of our row sorting algorithm, suppose we are given a set P of n arbitrary points $p_1, \ldots, p_n$ and that we want to find all triples (p_i, p_j, p_k) such that p_i, p_j, and p_k form an isosceles triangle. Equivalently, we want to find for each p_i all pairs (p_j, p_k) such that $\mathrm{d}(p_i, p_j) = \mathrm{d}(p_i, p_k)$. If for each p_i, we have the other points sorted by distance from p_i, then a simple linear scan of the sorted list for p_i gives us all the pairs of points that are equidistant from p_i. Thus, sorting P with respect to itself in $O(n^2(1 + \lg \ell))$ time allows us to find all of the isosceles triangles formed by points of P in $O(n^2(1 + \lg \ell))$ time. Similarly, if the points of P are the vertices of a convex n-gon in counterclockwise order, then we can find all of the isosceles triangles formed by points of P in $O(n^2)$ time. (Note that in Section 1, we made the simplifying assumption that no two entries of a totally monotone array are equal. However, as was also mentioned in Section 1, all of the algorithms and analyses in this paper are easily modified to handle equalities.)

5 Array Sorting

As a final variation on our paper's theme, we consider the problem of sorting all the entries of an $m \times n$ totally monotone array. A primary motivation for considering this problem is the following problem from computational geometry: given n points $p_1, \ldots, p_n$ in the plane, sort the $\binom{n}{2}$ distances $\mathrm{d}(p_i, p_j)$, corresponding to all pairs (p_i, p_j) of points. If distance is measured in terms of the L_1 metric (*i.e.*, for $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$, $\mathrm{d}(p_i, p_j) = |x_i - x_j| + |y_i - y_j|$), then this problem can be solved in $O(n^2)$ time [Fre76]. However, if distance is

measured in terms of the L_2 (Euclidean) metric, then no $o(n^2 \lg n)$ time algorithm for this problem is known. For the special case of this problem when $p_1, \ldots, p_n$ are the vertices of a convex polygon in counterclockwise order, the distance array defined in Section 1 gives a reduction to the array sorting problem for a totally monotone array. Unfortunately, unlike the three array problems we considered in Sections 2–4, the array sorting problem is not significantly easier than the general problem of sorting mn arbitrary values, which takes $\Theta(mn \lg mn)$ time. Specifically, we can show a simple $\Omega(mn \lg m)$ time lower bound for the array sorting problem, which implies that the aforementioned distance array reduction does not help us in sorting the $\binom{n}{2}$ Euclidean distances associated with the vertices of a convex n-gon. Furthermore, since we can solve the array sorting problem in $O(mn \lg m)$ time by applying the row sorting algorithm of the previous section and then merging the m sorted rows, we have an optimal algorithm for the problem.

We prove the lower bound on the array sorting problem as follows. Let $L_1, \ldots, L_n$ denote n independent lists, each containing m positive integers, where for $1 \leq j \leq n$, $L_j = l_j[1], \ldots, l_j[m]$. Sorting any one of these lists requires $\Omega(m \lg m)$ time in the decision tree model; thus, sorting all of the lists requires $\Omega(mn \lg m)$ time. In $o(mn \lg m)$ time, we will reduce the problem of sorting $L_1, \ldots, L_n$ to an array sorting problem, thereby obtaining the desired lower bound on array sorting.

In $O(mn)$ time, we can compute the maximum integer l^* in any of the lists. Let $M = l^* + 1$. Now consider the $m \times n$ array $A = \{a[i,j]\}$, where $a[i,j] = jM + l_j[i]$. A is clearly totally monotone, since for all i, $a[i,1] < a[i,2] < \cdots < a[i,n-1] < a[i,n]$. Furthermore, the rank of $a[i,j]$ in A is $(j-1)M$ plus the rank of $l_j[i]$ in L_j. Thus, sorting the entries of A implicitly sorts $L_1, \ldots, L_n$.

Using a similar but slightly more complicated construction, we have also been able to obtain an $\Omega(mn \lg m)$ lower bound on the time required to sort the entries of an $m \times n$ array A, such that $m \leq n$ and both A and its transpose are totally monotone. However, for the sake of brevity, we omit the proof of this lower bound.

6 Conclusion

In this paper, we explore two fundamental comparison problems, selection and sorting, in the context of totally monotone arrays. We provide simple but efficient algorithms for two selection problems and a sorting problem involving totally monotone arrays, algorithms that take advantage of an array's total monotonicity to obtain significantly better results than are possible for arbitrary arrays. We also present several applications of these algorithms to problems in computational geometry. We leave the following important questions unresolved:

1. In Section 3, we give an algorithm for computing the k largest entries overall in an array A such that both A and its transpose are totally monotone. It remains open whether a comparable result can be obtained for totally monotone arrays whose transposes are not totally monotone.

2. The only array problem considered in this paper for which we obtain matching upper and lower bounds is the array sorting problem discussed in Section 5. It remains open whether the algorithms for row selection, array selection, and row sorting given in Sections 2–4 can be improved or nontrivial lower bounds for these problems obtained. (Lower bounds might follow from the sizes of the various problems' search spaces — for example, a lower bound of $\Omega(S)$ on the number of combinations of row permutations possible for a totally monotone array would imply an $\Omega(\lg S)$ lower bound on the time necessary to sort the array's rows in a linear decision tree model.)

3. In Subsection 4.2, we applied our algorithm for sorting the rows of a totally monotone array to the vertices of a convex polygon P, obtaining for each vertex v an ordering of the other vertices of P by distance from v. We then extended this technique to arbitrary point sets. However, it remains open whether our two selection algorithms for totally monotone arrays, which we also apply to the vertices of a convex polygon, can likewise be applied to arbitrary point sets.

Acknowledgements

The authors are grateful to Alok Aggarwal, who first suggested we consider selection and sorting in totally monotone arrays, brought [Cha87] and [Fre76] to our attention, and provided a number of helpful comments on an early draft of this paper. The authors also thank Michelangelo Grigni for bringing [FJ82] to our attention. Portions of this paper have appeared previously in Dina Kravets' S.M. thesis[Kra88].

References

[AGSS87] A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor. A linear time algorithm for computing the Voronoi diagram of a convex polygon. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 39–47, 1987.

[AK89] A. Aggarwal and D. Kravets. A linear time algorithm for finding all farthest neighbors in a convex polygon. *Information Processing Letters*, 31:17–20, 1989.

[AKM+87] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[AP89a] A. Aggarwal and J. Park. Parallel searching in multidimensional monotone arrays. *Journal of Algorithms*, 1989. Submitted. Portions of this paper appear in *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, 1988.

[AP89b] A. Aggarwal and J. Park. Sequential searching in multidimensional monotone arrays. *Journal of Algorithms*, 1989. Submitted. Portions of this paper appear in *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, 1988.

[BFP+72] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1972.

[Cha85] B. M. Chazelle. On the convex layers of a planar set. *IEEE Transactions on Information Theory*, IT–31:509–517, 1985.

[Cha87] B. M. Chazelle. Some techniques for geometric searching with implicit set representations. *Acta Informatica*, 24:565–582, 1987.

[Ede86] H. Edelsbrunner. Edge skeletons in arrangements with applications. *Algorithmica*, 1:93–109, 1986.

[FJ82] G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in X + Y and matrices with sorted columns. *Journal of Computer and System Sciences*, 24:197–208, 1982.

[Fre76] M. L. Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1:355–361, 1976.

[Gui88] L. J. Guibas, 1988. Personal communication (via Alok Aggarwal).

[Knu73] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[Kra88] D. Kravets. Finding farthest neighbors in a convex polygon and related problems. Master's thesis, Massachusetts Institute of Technology, December 1988. Also published as Technical Report MIT/LCS/TR-437, January, 1989.

[Lee82] D. T. Lee. On k-nearest neighbor Voronoi diagrams in the plane. *IEEE Transactions on Computers*, C–31:478–487, 1982.

[LP78] D. T. Lee and F. P. Preparata. The all nearest-neighbor problem for convex polygons. *Information Processing Letters*, 7(4):189–192, 1978.

[MSS89] Y. Mansour, B. Schieber, and S. Sen, September 1989. Personal communication (via Yishay Mansour).

[Wil88] R. Wilber. The concave least-weight subsequence problem revisited. *Journal of Algorithms*, 9:418–425, 1988.

[YL79] C. C. Yang and D. T. Lee. A note on the all nearest-neighbor problem for convex polygons. *Information Processing Letters*, 8(4):193–194, 1979.