SIAM J. COMPUT.
Vol. 15, No. 1, February 1986

022

COMPUTING THE LARGEST EMPTY RECTANGLE*

B. CHAZELLE†, R. L. DRYSDALE‡ AND D. T. LEE§

Abstract. We consider the following problem: Given a rectangle containing N points, find the largest area subrectangle with sides parallel to those of the original rectangle which contains none of the given points. If the rectangle is a piece of fabric or sheet metal and the points are flaws, this problem is finding the largest-area rectangular piece which can be salvaged. A previously known result [13] takes $O(N^2)$ worst-case and $O(N \log^2 N)$ expected time. This paper presents an $O(N \log^3 N)$ time, $O(N \log N)$ space algorithm to solve this problem. It uses a divide-and-conquer approach similar to the ones used by Bentley [1] and introduces a new notion of Voronoi diagram along with a method for efficient computation of certain functions over paths of a tree.

Key words. computational geometry, divide-and-conquer, free tree, location theory, optimization

1. Introduction. We consider the following problem: Given a rectangle containing n points, find the largest area subrectangle with sides parallel to those of the original rectangle which contains none of the given points. If the rectangle is a piece of fabric or sheet metal and the points are flaws, this problem is finding the largest-area rectangular piece which can be salvaged. The special case in which a largest empty square is desired has been solved in $O(n \log n)$ time using Voronoi diagrams in L_1-(L_∞-)metric [7], [12], which is just a variation of the largest empty circle problem studied by Shamos [14], [15]. In [13] an $O(n^2)$ worst-case and $O(n \log^2 n)$ expected-time algorithm is presented for the largest empty rectangle problem. Other related problems can be found in [3], [5].

This paper presents an $O(n \log^3 n)$ time, $O(n \log n)$ space algorithm to solve this problem. It uses a divide-and-conquer approach similar to the ones used by Bentley [1].

2. General approach. We first note that the largest area rectangle with sides parallel to the bounding rectangle will have each edge supported by either an edge of the bounding rectangle or by at least one of the given points. (If the set of points contains two or more points lying on a vertical or horizontal line, an edge of the largest rectangle may be supported by more than one point.) Any rectangle is uniquely determined by its four supports (points or edges of the bounding rectangle). Therefore a naive algorithm could choose quadruples of support and then test to see if any points lie inside the rectangle formed. This method requires $O(n^5)$ time. However, it is shown in [13] that the number of such empty rectangles is at most $O(n^2)$ and that by carefully maintaining those rectangles the one with the largest area can be found in $O(n^2)$ time.

We shall in this paper present a divide-and-conquer algorithm. Let $p_1, p_2, \cdots, p_n$ be the n points sorted by x-coordinate and $x_{\min}$, $x_{\max}$, $y_{\min}$, and $y_{\max}$ be the boundaries of the bounding rectangle. Let the coordinates of point p_i be (x_i, y_i). Our algorithm splits the points into two halves by x-coordinate. We recursively solve the problem for the sets $S_1 = \{p_1, \cdots, p_{\lfloor n/2 \rfloor}\}$ and $S_2 = \{p_{\lfloor n/2 \rfloor + 1}, \cdots, p_n\}$. (The bounding rectangles of these recursive calls must be adjusted. The right boundary for the left call is $x_{\lfloor n/2 \rfloor}$ and the left boundary for the right call is $x_{\lfloor n/2 \rfloor + 1}$.) These calls determine the largest

* Received by the editors July 7, 1983, and in revised form September 29, 1984. This research was supported in part by the National Science Foundation under grants MCS 8202359, and ECS 8121741.

† Department of Computer Science, Brown University, Providence, Rhode Island 02912.

‡ Department of Mathematics and Computer Science, Dartmouth College, Hanover, New Hampshire 03755.

§ Department of Electrical Engineering/Computer Science, Northwestern University, Evanston, Illinois 60201.

rectangles with all four supporting points or edges in one half or the other. What remains are rectangles with supports in both halves. These rectangles contain either three supports in one half and one support in the other or two supports in each half (see Fig. 1). Our algorithm finds the largest rectangle of each type, and then returns the largest rectangle found with either all four supports in one half, three supports in one half and one in the other, or two on each side as the largest rectangle.

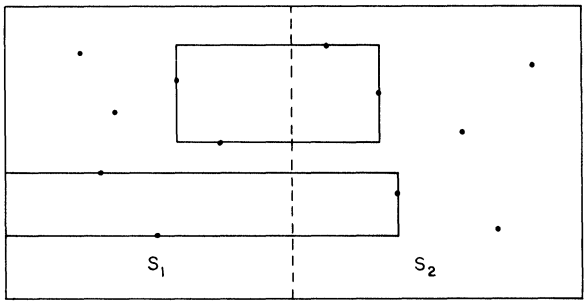

FIG. 1. *Possible empty rectangles.*

Therefore the run time of our algorithm is governed by the time required to find the largest rectangle with three supports in one half and one in the other and the time to find the largest rectangle with two supports in each half. That is, we have

$$T(N) \leqq 2T(N/2) + C(N) + D(N), \tag{1}$$

where $T(N)$ denotes the run time of the algorithm for the largest empty rectangle problem for N points, $C(N)$ the time for finding the largest empty rectangle with three supports in one half and one support in the other half and $D(N)$ the time for finding the rectangle with two supports in each half. As will be shown later, $C(N) = O(N)$ and $D(N) = O(N \log^2 N)$, which gives $T(N) = O(N \log^3 N)$.

3. Three supports in one half, one in the other. This is the easier of the two subproblems. We will look at the case of three supports in the left half and one in the right and present a linear time algorithm for this case. The other case is symmetrical, and is solved the same way.

Our first observation is that we need only consider approximately n rectangles. If the left support is a given point p_j, the rectangle is completely determined. The upper edge is supported by the first point above p_j which also lies to its right. If no such point exists, the rectangle is supported by the top edge of the bounding rectangle. Graphically, this support is found by drawing a horizontal ray to the right from p_j and sweeping it upward until it encounters either a point in the left half or the top edge of the bounding rectangle. Similarly the bottom edge is supported by the first point below p_j which also lies to its right if such a point exists, and the bottom edge of the bounding rectangle otherwise. These are the only top and bottom supports possible if all three supports are to lie in the left half. Let upper_j and lower_j denote, respectively, the upper and lower supports of the rectangle whose left support is p_j. The right support right_j is found by extending the rectangle supported above by upper_j and below by lower_j to the right until it encounters either a point in the right half or the right edge of the bounding rectangle. (Note that if several points in the left half life on a horizontal line, only the rightmost will support the left side of a rectangle.)

There are $\lfloor n/2 \rfloor + 1$ rectangles supported by the left edge of the bounding rectangle. If the p_j are sorted from top to bottom there is one rectangle above the top point, one

between each pair of points, and one below the bottom point. (If two or more points lie on a horizontal line, then some of these rectangles will be empty.)

A naive algorithm would take each point in the left and find the upper and lower supports of its rectangle, using $O(n)$ time for each. Finding the right support would take $O(n)$ additional time. Similarly, the right support of each rectangle supported by the left side of the bounding rectangle could be found in $O(n)$ time, so the whole process would take $O(n^2)$ time. However, we can find the largest rectangle in $O(n)$ time given the points sorted by y-coordinate.

Finding the upper and lower supports of rectangles with left support at the left side of the bounding rectangle is trivial if the points are sorted by y-coordinate. We present below a linear time algorithm to find the upper, lower and right supports of each rectangle supported on the left by a point p_j in the left half. Suppose that the points in the left half are sorted from top to bottom as $p_1, p_2, \cdots, p_m$. gap_j is defined to be the right support of the rectangle supported above by p_{j-1} and below by p_j, with $p_0 = (x_{\max}, y_{\max})$ and $p_{m+1} = (x_{\max}, y_{\min})$. $\text{gap}_1, \text{gap}_2, \cdots, \text{gap}_m$ are obtained in linear time with the points presorted by y-coordinate. Right_j is initialized to be the leftmost point in the right half with y-coordinate y_j, or $q_j = (x'_{\max}, y_j)$ if no such point exists, where $x'_{\max}$ is the right boundary of the current bounding rectangle for the right half. The arrays *above* and *below* are used to hold the points with running minimum x-coordinates (see Fig. 2).

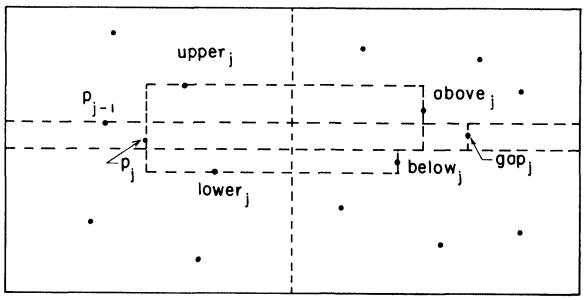

FIG. 2. *Illustration of supports of a point* p_j.

Our algorithm uses a stack. It takes advantage of the fact that when processing the points from top to bottom, the upper support of a point is the first point lying above it which also lies to its right. Therefore, points lying above the current point but to its left will never be upper supports for subsequent points and can be eliminated. A symmetric argument can be made about lower supports for which we process the points from bottom to top. This gives rise to the following algorithm.

1. Initialize the stack with upper support p_0 and $q_0 = (x'_{\max}, y_{\max})$ so that top is p_0, $x(\text{top}) = x_{\max}$, $y(\text{top}) = y_{\max}$, above (top) $= q_0$ and $x(q_0) = x'_{\max}$.
2. Scan the points $p_1, p_2, \cdots, p_m$ from top to bottom. For each point p_j encountered we do the following:
   2.1. If $x(\text{gap}(j)) < x(\text{right}(j))$ then above (j) is set to gap (j) and to right (j) otherwise.
   2.2. While $x(\text{top}) \leqq x(j)$ do;
   if $x(\text{above}(j)) \geqq x(\text{above (top)})$ then above (j) is set to above (top);
   pop the stack.
   2.3. upper (j) is set to top.
   2.4. Push p_j onto the stack.

3. Reinitialize the stack with lower support p_{m+1} and $q_{m+1}=(x'_{\max}, y_{\min})$ so that top is p_{m+1}, $x(\text{top})=x_{\max}$, $y(\text{top})=y_{\min}$, below (top) $=q_{m+1}$ and $x(q_{m+1})=x'_{\max}$.
4. Scan the points $p_1, p_2, \cdots, p_m$ from bottom to top. For each point p_j encountered we do the following:
   4.1. If $x(\text{gap}(j+1))<x(\text{right}(j))$ then below (j) is set to gap $(j+1)$ and to right (j) otherwise.
   4.2. While $x(\text{top})\leqq x(j)$ do;
   if $x(\text{below}(j))\geqq x(\text{below (top)})$ then below (j) is set to below (top);
   pop the stack.
   4.3. lower (j) is set to top.
   4.4. Push p_j onto the stack.
5. For each point p_j, $j=1, 2, \cdots, m$ do;
   if $x(\text{above}(j))<x(\text{below}(j))$
   then right (j) is set to above (j)
   else right (j) is set to below (j).

As can be easily shown, the algorithm examines each candidate left support (steps 2, 4 and 5) once, taking a total of $O(m)$ time. So we conclude this section with the following.

LEMMA 1. *The time $C(N)$ for finding the largest empty rectangle for N points with three supports in one half and one support in the other half is $O(N)$.*

4. Two supports in each half. Notice that the two supports must be on adjacent sides of the rectangle. Namely, the two supports in the left half must determine either the upper left corner or the lower left corner of the rectangle and the other two supports in the right half determine the lower right corner or the upper right corner of the rectangle respectively. Since these two cases are similar, we shall consider only the case where the two supports in the left half determine the lower left corner and the two supports in the right half determine the upper right corner of the rectangle. If we can identify all the possible lower left corner points in the left half and all the possible upper right corner points in the right half, then what remains to be solved is to find the so-called *largest empty corner rectangle* (LECR) which is determined by a corner point in each half. Therefore, we shall first compute all the possible corner points in each half and then devote ourselves to the problem of finding the largest empty corner rectangle.

4.1. Computation of corner points. We observe that two points p_i and p_j determine the lower left corner point of an empty rectangle iff p_j is lower$_i$. Thus, the point LC_i, $i=1, 2, \cdots, m$, determined by p_i and lower$_i$ is a lower left corner point and has as its x- and y-coordinates equal to x_i and $y(\text{lower}_i)$ respectively. In addition to these lower left corner points we include the points $L_i=(x_{\min}, y_i)$, $i=1, 2, \cdots, m$, i.e., the points on the left boundary, and the original set of points to form the set $CL=\{LC_1, LC_2, \cdots, LC_s\}$ where $s\leqq 3m$. All the possible upper right corner points in the right half can be computed in an analogous manner. We now have two sets of corner points $CL=\{LC_1, LC_2, \cdots, LC_s\}$ and $CR=\{RC_1, RC_2, \cdots, RC_t\}$ and want to find the largest empty (corner) rectangle whose lower left corner and upper right corner are from CL and CR respectively. Figure 3 shows the corner points in each half, with ● and × representing given and newly created points, respectively. Before we give the algorithm for finding the LECR, some observations are in order. Notice that not every point in CL can be paired with a point in CR. The empty rectangle that we seek must be a rectangle with *exactly two supports in each half.* For example, in Fig. 3 the point

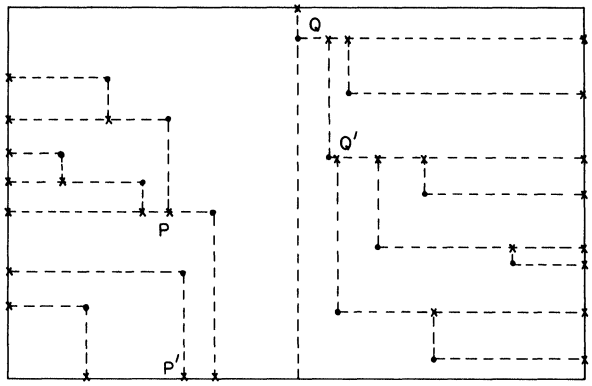

FIG. 3. *Newly created corner points.*

P in CL can only be paired with point Q in CR (but not Q'). Specifically, the following *pairing condition* must be satisfied: the corner point LC_i with left support p_l and bottom support p_b can only be paired with a point RC_j whose corresponding top support is *higher* than p_l and right support *higher* than p_b. Secondly, since original points are included in CL and CR, we should not use any of these points as corner points of the corner rectangle. However, as we will see in the following two lemmas, these problems will not arise as far as the computation of the LECR is concerned. Inclusion of the given points in CL and CR is to ensure that the corner rectangle thus determined contains no given points in its interior.

LEMMA 2. *The largest empty corner rectangle cannot use any of the given points as corner points. Furthermore, the corner points from CL and CR, respectively, satisfy the pairing condition prescribed above.*

Proof. Let the LECR be determined by LC_i and RC_j for some i and j. Suppose that LC_i is one of the given points in the left half. Since there exists in CL a point LC_k to the left of LC_i which is the corner point determined by some point p and LC_i, where $x(LC_k) = x(p)$ and $y(LC_k) = y(LC_i)$, and LC_k can be paired with RC_j to form a larger empty corner rectangle, we have a contradiction. On the other hand, if LC_i violates the pairing condition that the associated left support is higher than RC_j, then we can always find another corner point $LC_{k'}$ in CL with $y(LC_{k'}) = y(LC_i)$ that satisfies the pairing condition and can be paired with RC_j to form a larger empty corner rectangle, a contradiction. The case where RC_j is one of the given points in the right half or it violates the pairing condition that the associated right support is lower than LC_i can be handled in a similar way. This completes the proof.

Thus, the largest empty corner rectangle must use the newly created points as its corner points. We note that we only require that the corner rectangle contain none of the given points, so it may contain some of the newly created corner points in its interior. However, the following lemma rules out this possibility.

LEMMA 3. *If a corner rectangle does not contain any given point in its interior, then it must also not contain any newly created corner points.*

Proof. Suppose it contains a newly created left corner point LC in its interior. Let p_i and p_j be the two points in the left half that determine LC so that LC and p_j have the same y-coordinate. Since the corner rectangle is determined by a lower left corner and an upper right corner points that are in the left and right halves respectively, it must contain point p_j in its interior as well, a contradiction. The case where it contains a newly created right corner point in its interior can be handled similarly.

With the above two lemmas we can proceed to find a largest corner rectangle which is determined by a point in CL and a point in CR and which is "empty" in the sense that it does not contain any point (including those newly created corner points) in its interior.

4.2. Computing the largest empty corner rectangle. We first assume that the points in CL and in CR have been sorted in both x- and y-coordinates. Divide the sets CL and CR each into two subsets CL_1, CL_2 and CR_1, CR_2, respectively, with CL_1 above CL_2 and CR_1 above CR_2, using a horizontal line such that $CL_1 \cup CR_1$ is approximately of the same size as $CL_2 \cup CR_2$ (Fig. 4). Assume recursively that we have computed the LECR in $CL_1 \cup CR_1$ and in $CL_2 \cup CR_2$. So we may concentrate on the case where the lower left corner is in CL_2 and upper right corner is in CR_1. If $E(N)$ denotes the time complexity of the latter problem and $D(N)$ denotes that of the former problem, we have

$$D(N) \leqq 2D(N/2) + E(N). \tag{2}$$

Our first observation is that we may discard all the points of CR_1 that "dominate" any other. (A point p is said to dominate point q if both p's x- and y-coordinates are greater than those of q's; and a point is maximal if it is not dominated by any other point.) This is identical to keeping the maxima of the mirror image of CR_1 with respect to the origin, so it can be accomplished in linear time, since the points of CR_1 are sorted in x-order. For points with the same y-coordinate we further trim them by keeping only the rightmost one that does not dominate any other point in CR_1. Similarly, for points with the same x-coordinate we keep only the topmost point that does not dominate any other point in CR_1. See Fig. 4, in which points eliminated are marked

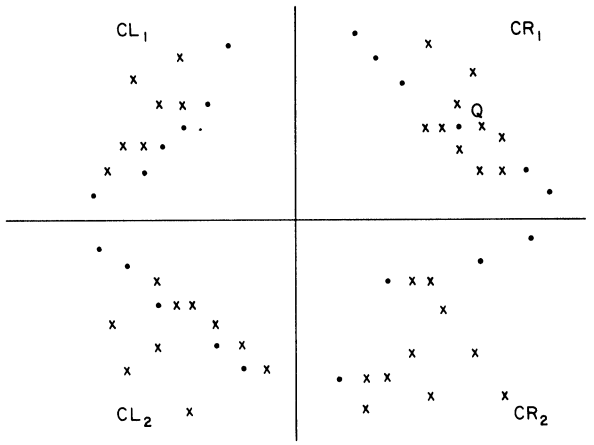

FIG. 4. *Subdivision of the points into four subsets and the trimming operation.*

as $\times$. The reason why they can be trimmed is that they cannot form the LECR with a point in CL_2. Similar remarks can be made about CL_2, and we can trim this set in a similar way. Finally, we can also apply this clean-up to CL_1 and CR_2. Note that this final clean-up removes from CL_1 the points that have the same y-coordinate except the rightmost one and removes from CR_2 the points that have the same y-coordinate except the leftmost one. This procedure can be accomplished in $O(N)$ time. In what follows we assume that these sets have been trimmed.

The next step is to determine, for each point in CL_2, the set of points in CR_1 with which the point can be paired. This set is clearly a contiguous subsequence of the

points $M_1, M_2, \cdots, M_u$ of CR_1 given in ascending x-order. We will therefore precompute the functions $l(P)$, and $r(P)$ such that the set $\{M_{l(P)}, M_{l(P)+1}, \cdots, M_{r(P)}\}$ contains exactly the points of CR_1 which can be paired with P to form an empty corner rectangle (Fig. 5). It is easy to precompute the function l (and by the same token, r) in linear time, proceeding as follows: assume that each set CL_1, CL_2, CR_1 and CR_2 has been sorted by x-coordinates. In a merge-like scan through CL_1 and CL_2, we compute, for each point P of CL_2, its "counterpart" in CL_1, i.e., the leftmost point of CL_1 to the right of the vertical line passing through P. We perform the same operation with respect to CL_1 and CR_1 (rotated by 90 degrees), and the conjunction of the two lists thus obtained precisely provides the desired correspondence between P and $M_{l(P)}$. We compute the function $r(P)$ similarly.

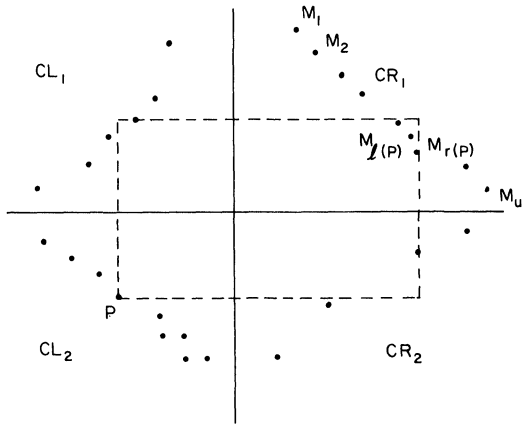

FIG. 5. *The points in CR_1 that can be paired with P in CL_2.*

Once the set $\{M_{l(P)}, M_{l(P)+1}, \cdots, M_{r(P)}\}$ associated with P is computed, in order to facilitate searching of a point M_i in the set with which to pair P to form the LECR we make use of the notion of so-called LL-diagram (Lower-Left-diagram), which is similar to the notion of Voronoi diagram (see, for example, [10], [11], [15]).

Computing the LL-diagram. The LL-diagram of a set $S = \{M_1, M_2, \cdots, M_N\}$, denoted $LL(S)$, is defined as follows: $LL(S)$ is a set of regions $\{V(M_1), V(M_2), \cdots, V(M_N)\}$, where

$$V(M_i) = \{M \in NE^* \mid d(M, M_i) \geqq d(M, M_j) \text{ for all } j = 1, 2, \cdots, N\}$$

and $d(A, B)$, the d-distance between A and B, measures the area of the corner rectangle between A and B if B dominates A and is zero otherwise; NE^* denotes the region $(-\infty, x_{\max}] \times (-\infty, y_{\max}]$ excluding the smallest enclosing rectangle of S, where $x_{\max}$ and $y_{\max}$ are maximum x- and y-coordinates of S respectively, and is the crossed-line area shown in Fig. 6. Note that if a point M of S is dominated by another, its associated region $V(M)$ is empty. Thus, we only consider the case where S contains only maxima. The LL-diagram of S has the following properties (Lemmas 4, 5 and 6).

LEMMA 4. *Let S be a set of N maxima, $M_1, M_2, \cdots, M_N$. Then $LL(S)$ consists of a set of possibly unbounded polygons which partitions NE^*. All the polygons (except one) are convex, and $LL(S)$ involves only $O(N)$ edges.*

Proof. First consider the case where S consists of only two points $A(0, v)$ and $B(u, 0)$ with u and v positive. The points $M(x, y)$ in NE^* farther from A than from B (with respect to d) satisfy the relation $x(v-y) \leqq y(u-x)$. This reduces to $y \geqq 0$ or $vx - uy \leqq 0$, which is the area of NE^* above the line passing by the diagonal (other

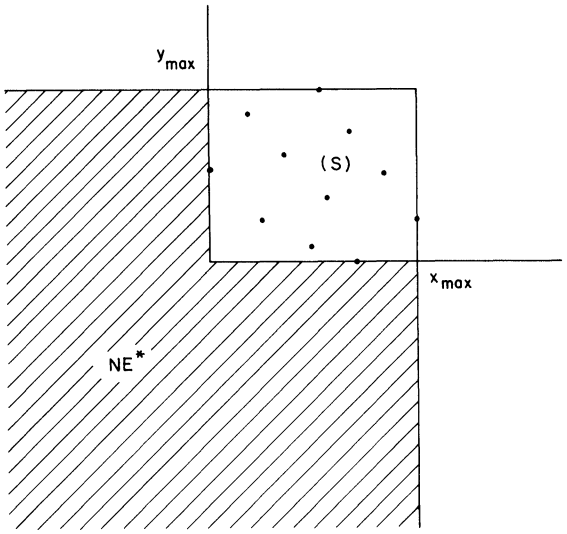

FIG. 6. *Domain of definition of LL-diagrams.*

than AB) of the corner rectangle determined by A and B (Fig. 7). In general, consider the intersection W of the regions associated with M_i in $LL(M_k, M_i)$, $k = 1, 2, \cdots, N$. Since W is the intersection of unbounded convex polygons, it is itself a possibly unbounded convex polygon. Also, since the domain of definition, NE^*, as defined for each $LL(M_k, M_i)$, contains the domain of definition for $LL(S)$, the region $V(M_i)$ is simply the intersection of W with NE^*; it is therefore a possibly unbounded polygon, which is always convex, except for the polygon which contains the "corner" of NE^*. We can also see that $V(M_i)$ has an edge on the boundary of NE^*. (Note that none of the regions associated with M_i in $LL(M_k, M_i)$ lies strictly inside NE^*.) The last point to make is that since $LL(S)$ is a planar graph with $O(N)$ faces, all of whose vertices have degree $\geqq 3$, it involves $O(N)$ vertices.

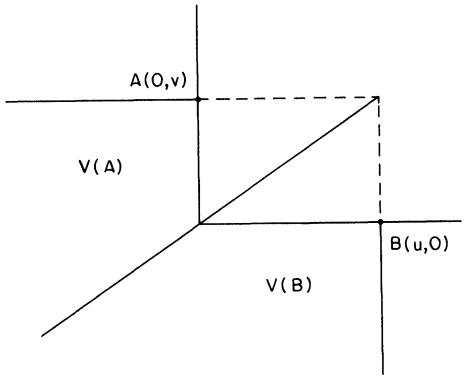

FIG. 7. *LL-diagram for points A and B.*

LEMMA 5. *Let $M_1, M_2, \cdots, M_N$ occur in this order with ascending x-coordinate and let L be any line parallel to the x-axis. It is impossible to find two points A and B on L in NE^* with increasing coordinates such that A and B lie in $V(M_i)$ and $V(M_j)$, respectively, with $i > j$* (Fig. 8).

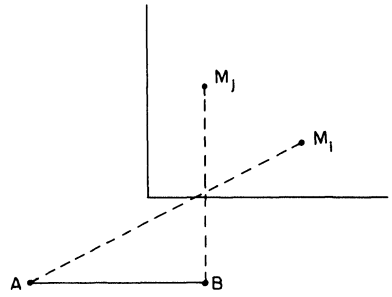

FIG. 8. *Point A cannot be in $V(M_i)$ and point B in $V(M_j)$.*

Proof. Immediate from the definition of $LL(M_i, M_j)$.

We now proceed with the computation of $LL(S)$ using a standard divide-and-conquer technique. We sort the points in S in ascending x-coordinates as $M_1, M_2, \cdots, M_N$. Recursively compute $L_1 = LL(S_1)$ and $L_2 = LL(S_2)$, where $S_1 = \{M_1, M_2, \cdots, M_{\lfloor N/2 \rfloor}\}$ and $S_2 = \{M_{\lfloor N/2 \rfloor + 1}, \cdots, M_N\}$, and then merge them. We start at the lower left corner of the corner rectangle determined by $M_{\lfloor N/2 \rfloor}$ and $M_{\lfloor N/2 \rfloor + 1}$ drawing the diagonal AB (Fig. 9a) downwards until we hit an edge of L_1 or L_2, at which point we *stitch* the two segments. We illustrate the stitching operation in Fig. 9b. Let us call a *diagonal* of a segment $\overline{M_iM_j}$, the line supporting the diagonal of the corner rectangle determined by M_i and M_j (other than $\overline{M_iM_j}$). Suppose that the line currently drawn follows the diagonal of $\overline{M_iM_j}$ downwards. When we encounter the diagonal of $\overline{M_jM_k}$, we cut through it and replace it by the diagonal of $\overline{M_iM_k}$ (Fig. 9b). By doing so, we recompute the LL-diagram of S locally around the path being thus drawn. If we iterate on this process i.e., drawing, hitting and stitching, we will produce a path **z**, which is obviously monotone with respect to both the x and y axes. This is

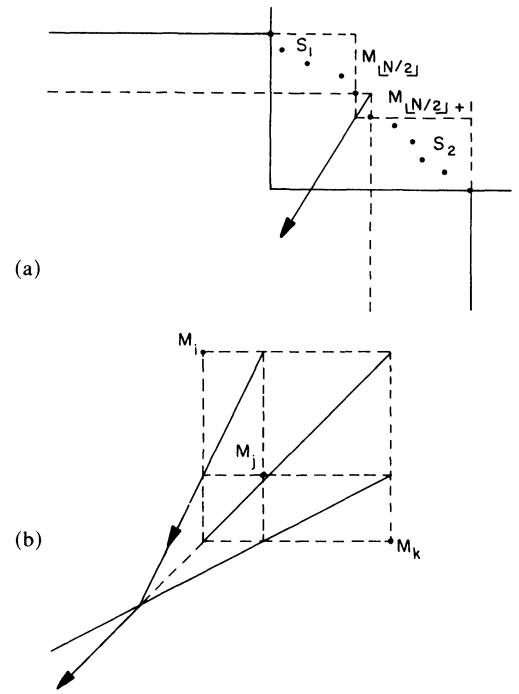

FIG. 9. *Stitching operation of two LL-diagrams.*

due to the fact that z is made of chunks of diagonal, which all have positive slopes. To ensure that stitching the two *LL*-diagrams L_1 and L_2 takes $O(N)$ time, we do the following. At all times, we keep track of the face in L_1 and the face in L_2 we are currently in. The stitching operation corresponds to leaving one face of, say L_1, for another face of L_1. At this point, we know the direction to follow, and we must compute the next hitting edge. To do so, we maintain a pointer p_1 (respectively p_2) to go around the current face of L_1 counterclockwise (respectively L_2 clockwise). Since all slopes are positive, pointers will always be descending; therefore we can move them in a simple round-robin fashion so as to detect the first intersection with the new drawing direction without ever backtracking. As usual, we remove all parts of L_1 and L_2 which have been cut and lie to the right and left, respectively, of the path z. We omit the details and conclude that the entire computation of $LL(S)$ takes $O(N \log N)$ time. Interested readers are referred to, for example, [10], [11], [15] for details of the merge concept.

LEMMA 6. *The LL-diagram of a set of N points can be computed in* $O(N \log N)$ *time.*

Proof. It suffices to show that in the recursive step of the above procedure each point to the left and to the right, respectively, of z has its farthest neighbor in S_1 and in S_2. Assume that this is not the case, and that, for example, a point M to the left of z has its farthest neighbor in S_2. Let us draw a horizontal line through M. This line will necessarily intersect the path z in exactly one point P. Let P^* be a point immediately to the left of z (Fig. 10). P is, by construction, in the region of a point of S_1, since we have already seen that the *LL*-diagram has been correctly constructed around z locally. This is a contradiction to Lemma 5, which completes the proof.

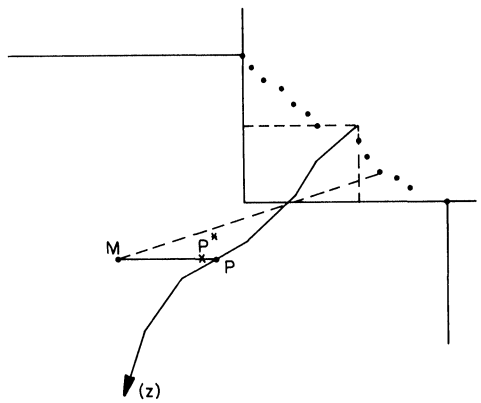

FIG. 10. *Illustration for the proof of Lemma* 6.

With the *LL*-diagrams in mind let us consider how they can be effectively used in finding the point of CR_1 to pair with each point in CL_1. Consider the complete binary tree T that has $M_1, M_2, \cdots, M_u$ of CR_1 for leaves, from left to right. Letting SL be the set of sequences of leaves of each subtree of T, we can use the standard segment-tree technique [2] to rewrite any contiguous subsequences of $M_1, M_2, \cdots, M_u$ as the concatenation of $O(\log u)$ sequences in SL (Fig. 11). It is clear that for any interval $[M_i, M_j]$ of consecutive leaves such decomposition can be obtained in $O(\log u)$ time by a simple search for M_i and M_j in T. We omit the details (see Bentley and Wood [2]). The purpose of this decomposition is to compute efficiently the farthest neighbor in $[M_{l(P)}, M_{r(P)}]$ of each point P in CL_2. To do so, we precompute the *LL*-diagram of each sequence of points in SL so that we may decompose $[M_{l(P)}, M_{r(P)}]$

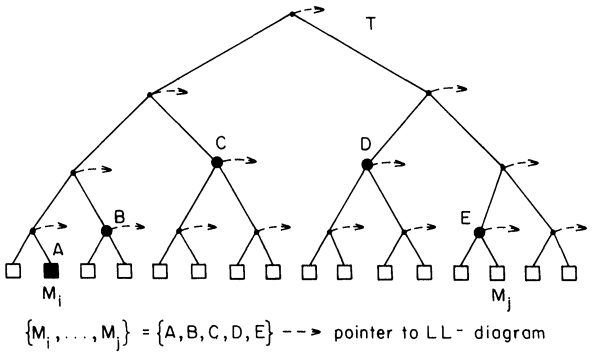

FIG. 11. *Decomposition of an interval* $[M_i, M_j]$ *into subintervals.*

into sequences in SL and search for the farthest neighbor of P in each of the sequences. A similar technique can be found in Gowda et al. [6]. Searching for the farthest neighbor is, in this case, equivalent to determining in which region of the LL-diagram P lies. This can be accomplished in logarithmic time, using only linear space with Kirkpatrick's planar point location algorithm [8].

To summarize, the preprocessing of the recursive step consists of:

1. Trimming CL_1, CL_2, CR_1 and CR_2 in $O(N)$ time and space.
2. Precomputing the functions l, and r in $O(N)$ time snd space.
3. Setting up the tree T for CR_1, and computing the LL-diagram of each sequence of points in SL. SL consists of one u-sequence, two $u/2$-sequences, $\cdots$, 2^k $u/2^k$-sequences, etc. It follows that computing all the LL-diagrams requires time $T(u) = 2T(u/2) + O(u) = O(u \log u)$ and space $S(u) = 2S(u/2) + O(u) = O(u \log u)$. (See Lemma 6 or [6].)
4. Setting up the preprocessing required by Kirkpatrick's planar point location algorithm for each LL-diagram computed, which requires $O(k)$ time and space for a set of k points.

Consequently, the total cost of the preprocessing amounts to $O(u \log u)$ time and space.

The computation of the farthest neighbor in $\{M_{l(P)}, \cdots, M_{r(P)}\}$ of each point P in CL_2 is done by performing $O(\log u)$ planar point searches, each requiring $O(\log u)$ time. Putting our results together, we conclude that it is possible to find the LECR with corners in CR_1 and CL_2 in time $E(N) = O(N \log^2 N)$ and space $O(N \log N)$. From the relations (1) and (2) we therefore have the following.

LEMMA 7. *The largest empty corner rectangle with corner points in CL and CR can be computed in* $D(N) = O(N \log^3 N)$ *time and* $O(N \log N)$ *space.*

THEOREM 1. *The largest empty rectangle problem for N points in the plane can be solved in* $T(N) = O(N \log^4 N)$ *time and* $O(N \log N)$ *space.*

4.3. An improved algorithm for computing LECR. The result of the previous section can still be improved using a less redundant representation. The redundancy comes partly from the horizontal recursion, since it is likely to entail repeated computations of the same LL-diagrams. Instead, we will set up a global data structure for the entire right half, namely the set CR.

Let $M_1, \cdots, M_u$ be the points of CR. We will arrange the points to be the nodes of a rooted tree T_{CR} that is constructed as follows. The root is an imaginary point situated entirely above and to the left of CR. First of all, the points with the same y-coordinates are connected to form chains, ordered in y-coordinate. Then for each chain we connect the leftmost point to the point in a higher chain that is directly above

it (Fig. 12). We note that if P is an arbitrary point in CL and M_i is the lowest (rightmost) point of CR higher than P, the path from M_i to the root of the tree contains the only points which can be paired with P to form an LECR (Fig. 12).

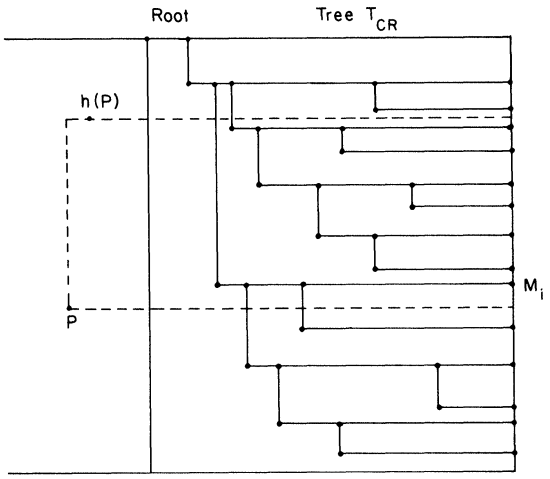

FIG. 12. *Construction of the tree* T_{CR}.

By analogy with the previous section, we will precompute for each point in CL the functions l and r, which will now point to nodes in T_{CR}. Notice that $M_{r(P)}$ is simply the lowest (rightmost) point of CR higher than P, and can be computed in linear time for each P in CL. Similarly we can compute the "vertical" obstacle $h(P) = \text{upper}(P)$, in CL for each point P in linear time. Next we precompute the function l as follows: consider each point P in descending y-coordinate and 1) if P is in CR, ensure that all the points and only the points on the path of T_{CR} from the root to P are arranged consecutively in a stack A, 2) if P is in CL, do a binary search in A in order to find the highest point below $h(P)$. Note that this point is exactly $M_{l(P)}$. Since T_{CR} is a tree, operation 1 consists simply of updating a stack, which will take a total of $O(N)$ time. Of course, each point of CL may cause an $O(\log N)$ search time. Therefore the total time complexity of the procedure is $O(N \log N)$.

Computing path functions in a free tree. Let T be a free tree with N vertices, each of degree at most 3. Recall that a free tree is a connected acyclic graph. We wish to compute "decomposable" functions over tree-paths very efficiently. We consider functions of the form $F(v, w)$, defined for any pair of vertices (v, w) of T. These functions are assumed to have the following property: there exists an associative operator OP computable in constant time, such that for any vertex z on the path from v to w, we have

$$F(v, w) = F(w, v) = OP(F(v, z), F(z, w)).$$

One trivial example of such a function is the distance between two nodes of the tree. For the application at hand, (v, w) will be a pair of the form $(l(P), r(P))$ and $F(v, w)$ will be the maximum d-distance between P and any point on the chain from $M_{l(P)}$ to $M_{r(P)}$.

Suppose now that in order to compute $F(v, w)$ we can use a data structure $L(v, w)$ so that $F(v, w)$ can then be evaluated in $O(f(N))$ time. We assume that $L(v, w)$ requires $O(t)$ space and can be computed in $O(t \log t)$ time, where t is the number of nodes on the path between v and w. Instead of precomputing all possible structures

$L(v, w)$ for all pairs of nodes (v, w), which would require $O(N^3)$ storage, we will compute only a subset of them $R = L(v_i, w_i)$, which takes only $O(N \log N)$ space, and has the property that the path between any pair of nodes in T is the concatenation of disjoint paths between $O(\log N)$ pairs (v_i, w_i). The availability of R clearly allows us to evaluate $F(v, w)$ in $O(f(N) \log N)$ time, assuming that we can express the path (v, w) as a sequence in R in $O(\log N)$ time.

The construction of R relies on the fact that it is possible in linear time to find a vertex (the centroid) of an N-vertex tree whose removal from the tree leaves subtrees with at most $N/2$ nodes [9]. We will compute R recursively. To do so, we will represent T, as a rooted ternary tree G defined as follows: let C, the centroid of T be the root of G. For the sake of simplicity we may assume without loss of generality that we have exactly 3 subtrees, T_1, T_2, T_3, rooted at C. We then proceed to compute their centroids, C_1, C_2, C_3, which we insert in G as the sons of the root C. We iterate on this process for each subtree, labeling the nodes in the following manner. Assume that T_i has exactly N_i vertices ($N_1 + N_2 + N_3 = N - 1$). The root C is labeled N and the general rule is that T_1 will be labeled recursively with the integers $\{1, \cdots, N_1\}$, T_2 with $\{N_1 + 1, \cdots, N_1 + N_2\}$, and T_3 with $\{N_1 + N_2 + 1, \cdots, N - 1\}$. To be consistent we will label the root of each subtree with the highest label available.

Our next task is to augment G with new edges, called *extra-edges*. For each vertex v of G we, in turn, apply the following procedure to all the vertices which are adjacent to v in T and are labeled lower than v in G. Let w be such a vertex. Since it clearly must lie in the subtree of G rooted at v, we link to v every node, including w, on the path in G from v to w, thus adding the so-called extra-edges. Figure 13a shows the labeling of the tree and Fig. 13b the ternary tree representation G. The dotted lines in Fig. 13b indicate the extra-edges that are introduced when the root node (labeled 26) is considered.

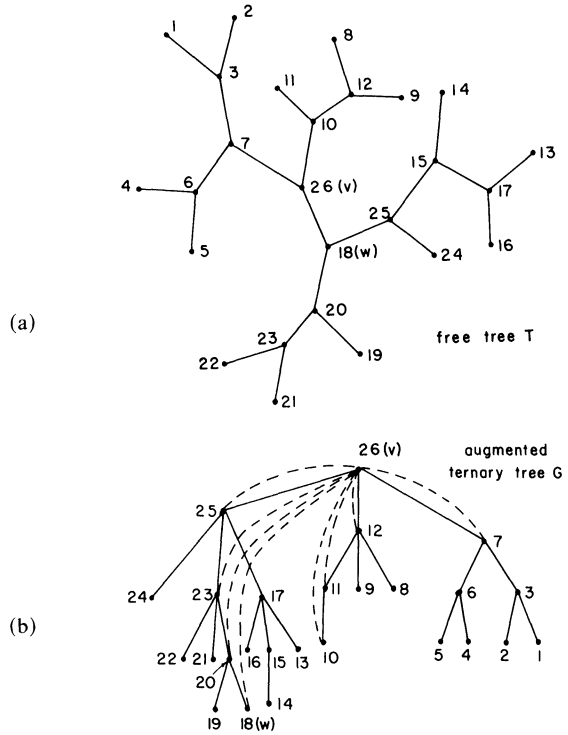

FIG. 13. *Converting a free tree to an augmented ternary tree.*

Since we can compute centroids in linear time and G clearly has height $O(\log N)$, it is easy to construct the tree augmented with extra-edges in time $\text{Max}(O(N \log N), H(N))$, where the first term accounts for the time needed for augmenting the extra-edges, and $H(N)$, the time for constructing G, is given by $H(N) = H(N_1) + H(N_2) + H(N_3) + O(N)$, $N_1 + N_2 + N_3 = N - 1$, and $N_1, N_2, N_3 \leqq N/2$, i.e., $H(N) = O(N \log N)$. The final addition to G is to set pointers from each edge (u, v) of G to the structure $L(u, v)$.

It is not difficult to evaluate the overall time and space complexities, $T(N)$ and $S(N)$, respectively, of the preprocessing we have just described. Since an edge in G from the root to a vertex k levels deeper gives rise to a path in T of length at most $N/2^k$, the time and space complexities, respectively, for computing all the structures of the form $L(\text{root}, x)$ required by our algorithm is $O(N \log N)$ and $O(N)$. (Recall that the structure $L(v, w)$ is assumed to take $O(t)$ space and $O(t \log t)$ time to construct with t being the number of nodes on the path in T from v to w.) Thus,

$$T(N) = T(N_1) + T(N_2) + T(N_3) + O(N \log N)$$

and

$$S(N) = S(N_1) + S(N_2) + S(N_3) + O(N)$$

with $N_1 + N_2 + N_3 = N - 1$, and $N_1, N_2, N_3 \leqq N/2$. This gives rise to the following complexity bounds for computing the preprocessing structure R: $T(N) = O(N \log^2 N)$ and $S(N) = O(N \log N)$.

We can now show that this preprocessing allows us to evaluate $F(u, v)$ for any pair (u, v) of nodes in T in time $O(f(N) \log N)$. To do so, we walk up the path in G from u towards the root, stopping at the first node w with a label exceeding that of v. We know that v lies in one of the subtrees of w. Note also that the labeling of G allows us to go down from w to v in time proportional to the length of the path. At this point we perform the same operations for u and v in turn, so we may describe the procedure for u only. Let $(w, u_1, \cdots, u_k)$ $(u_k = u)$ be the path in G from w to u. If this path has extra-edges connecting w to some u_i, we note that G must have a set of consecutive extra-edges $wu_1, wu_2, \cdots, wu_j$ between w and u. We collect the last extra-edges wu_j and iterate on this process, restarting at u_j (Fig. 14). If w does not have an extra-edge on its path to u, we simply collect the edges wu_1 and iterate from u_1. Note that this *collecting* operation takes only $O(\text{length of path from } w \text{ to } u) = O(\log N)$. Finally we compute $F(u, v)$ by evaluating $OP(\cdots, F(u_i, v_i), \cdots)$, where the (u_i, v_i) are all the edges collected by the above procedure.

To show the correctness of our method it suffices to notice that the path formed by the edges collected from u to w, along with the path collected similarly from w to v, gives an exact partition of the path in T from u to v. There again, the reader can supply an easy proof of the fact. We conclude as follows.

LEMMA 8. *With $O(N \log N)$-space, $O(N \log^2 N)$-time preprocessing, it is possible to evaluate $F(u, v)$ in $O(f(N) \log N)$ time for any pair of vertices (u, v) in T.*

Note that the idea of decomposing tree-paths into canonical pieces is one aspect of a general mapping principle between linear lists and trees developed in [4].

Computing the LECR *efficiently.* A simple application of the previous paragraph provides an improved algorithm for computing the LECR of the sets $S = CL \cup CR$. Clearly T is our tree T_{CR}, the structure $L(u, v)$ is the LL-diagram of the points of CR on the path between u and v, preprocessed so as to allow for Kirkpatrick's planar point location algorithm, the function $F(u, v)$ simply returns the regions in $L(u, v)$ where a given point P of CL lies; its complexity is therefore $f(N) = O(\log N)$. Finally,

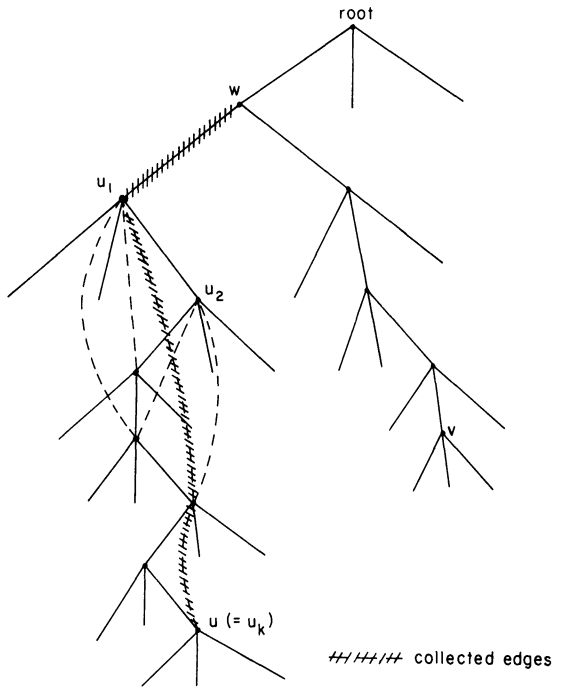

FIG. 14. *Collecting operation of a path from w to u.*

the operator $OP(M, M')$ returns the point (M or M') that forms the LECR with P. Note that the only discrepancy comes from the fact that T_{CR} is not necessarily a tree with degree $\leqq 3$. There is an easy fix, however. We simply introduce dummy vertices to reduce any excessive degree to 3. This adds only $O(N)$ vertices and thus does not affect the complexity of the algorithm. We are now in a position to compute the LECR in S. To do so, compute $F(M_{l(P)}, M_{r(P)})$ for all P in CL, and return the point which gives the largest area along with its upper right neighbor. We omit the details of this straightforward transformation. Thus, with Lemma 8 and the above discussion we have the following.

LEMMA 9. *It is possible to compute the* LECR *determined by a pair of points, each of which is in one of the sets CL and CR respectively in* $D(N) = O(N \log^2 N)$ *time and in* $O(N \log N)$ *space.*

Using relation (1) we can state our main result.

THEOREM 2. *The largest empty rectangle problem for a set of N points can be computed in* $O(N \log^3 N)$ *time and* $O(N \log N)$ *space.*

5. Conclusion. We have presented an $O(N \log^3 N)$-time and $O(N \log N)$ space algorithm for computing the largest area rectangle which contains none of the N points in its interior. A simpler version of the algorithm with running time $O(N \log^4 N)$ and space $O(N \log N)$ has also been given. The algorithms are primarily based on the divide-and-conquer strategy. We have addressed only the problem of locating a rectangle whose sides are parallel to those of the bounding rectangle of the given set of points. If the rectangle sought is arbitrarily oriented, the problem becomes much more difficult.

Naturally one may ask for an arbitrary polygon instead of a rectangle within a bounded region or generalize the problem to higher dimensions. We remark that if

the largest empty triangle is sought and the directions of the sides of the triangle have been predetermined, then the largest empty triangle can be found in $O(n \log n)$ time and $O(n)$ space using a divide-and-conquer techique similar to the one used in § 3.

REFERENCES

[1] J. L. BENTLEY, *Divide-and-conquer algorithms for closest point problems in multidimensional space*, Ph.D. thesis, Dept. Computer Science, Univ. North Carolina, Chapel Hill, NC, 1976.

[2] J. L. BENTLEY AND D. WOOD, *An optimal worst case algorithm for reporting intersections of rectangles*, IEEE Trans. Comput. (1980), pp. 571-577.

[3] J. E. BOYCE, D. P. DOBKIN, R. L. DRYSDALE III AND L. J. GUIBAS, *Finding extremal polygons*, Proc. ACM Symposium on Theory of Computing, 1982, pp. 282-289.

[4] B. M. CHAZELLE, *Computing on a free tree via complexity-preserving mappings*, Proc. 25th IEEE Symposium on Foundations of Computer Science, 1984, to appear.

[5] D. P. DOBKIN, R. L. DRYSDALE III AND L. J. GUIBAS, *Finding smallest polygons*, to appear in Advances of Computing Research, F. P. Preparata, ed., Jai Press.

[6] I. G. GOWDA, D. G. KIRKPATRICK, D. T. LEE AND A. NAAMAD, *Dynamic Voronoi diagrams*, IEEE Trans. Inform. Theory, IT-29 (1983), pp. 724-731.

[7] F. K. HWANG, *An $O(n \log n)$ algorithm for rectilinear minimal spanning trees*, J. ACM, 26 (1979), pp. 177-182.

[8] D. G. KIRKPATRICK, *Optimal search in planar subdivisions*, this Journal, 12 (Feb. 1983), pp. 28-35.

[9] D. E. KNUTH, *The Art of Computer Programming, Vol.* I: *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

[10] D. T. LEE, *Two dimensional Voronoi diagrams in the L_p-metric*, J. ACM, 27 (1980), pp. 604-618.

[11] D. T. LEE AND R. L. DRYSDALE III, *Generalization of Voronoi diagrams in the plane*, this Journal, 10 (1981), pp. 73-87.

[12] D. T. LEE AND C. K. WONG, *Voronoi diagrams in L_1-(L_∞-)metrics with 2-dimensional storage applications*, this Journal, 9 (1980), pp. 200-211.

[13] A. NAAMAD, W. L. HSU AND D. T. LEE, *On maximum empty rectangle problem*, Appl. Disc. Math., 8 (1984), pp. 267-277.

[14] M. I. SHAMOS, *Computational geometry*, Ph.D. dissertation, Dept. Computer Sciences, Yale Univ., New Haven, CT, 1978.

[15] M. I. SHAMOS AND D. HOEY, *Closest-point problem*, Proc. 16th IEEE Symposium on Foundations of Computer Science, 1975, pp. 151-162.