1984

A Note on Finding a Maximum Empty Rectangle

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Greg N. Frederickson
Purdue University, gnf@cs.purdue.edu

Report Number:
84-490

TR # 490

A NOTE ON FINDING A MAXIMUM EMPTY RECTANGLE

Mikhail J. Atallah

Greg N. Frederickson

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907.

Abstract

Given a rectangle A and a set S of n points in A, the *maximum empty rectangle* problem is that of finding a largest-area rectangle which is contained in A, has its sides parallel to those of A, and does not contain any of the points in S. This note describes an efficient algorithm for solving this problem.

1. Introduction

Given a rectangle A and a set S of n points in A, a *valid rectangle* is one which is contained in A, has its sides parallel to those of A, and does not contain any of the points in S. The *maximum empty rectangle* (MER) problem is that of finding the largest-area valid rectangle. This problem was first posed by Naamad, Lee and Hsu [NHS], who gave an $O(\min(n^2, s\log n))$ time algorithm for solving it. The parameter s is defined as the number of *restricted rectangles* of the problem, where a restricted rectangle (*RR* for short) is a valid rectangle such that each of its four edges either contains a point of S or coincides with an edge of A. Naamad et. al. prove that $s=O(n^2)$ and give an example in which $s=\Theta(n^2)$. They also show that when the points are drawn from a uniform distribution, the expected value of s is $O(n\log n)$, so that the average time

This work was supported in part by the Office of Naval Research under Contract N00014-84-K-0502.
This work was supported in part by the National Science Foundation under Grant DCR-8320124.

complexity of their algorithm is $O(n\log^2 n)$. More recently, an elegant algorithm of time complexity $O(n\log^3 n)$ has been given by Chazelle et. al. [CDL]. This is an improvement over [NLH] for large s, but is actually inferior in the average case. In this paper we describe an algorithm of time complexity $O(s+n\log^2 n)$. Thus its average time complexity matches that of [NLH], and its worst-case complexity improves on that of [NLH] in the range $n\log n < s < n^2$, and on that of [CDL] in the range $n \leq s \leq n\log^3 n$.

2. Preliminaries

We use $p_1, \ldots, p_n$ to denote the n input points, and $X(p_i), Y(p_i)$ to denote the coordinates of p_i. To simplify the exposition, we assume that no two points have same x–coordinate, and similarly for y–coordinates. Throughout, s denotes the total number of RR's.

Let Q be an RR. Q is *bounded from above (resp. below, left, right)* by p_i iff its top (resp. bottom, left, right) edge contains p_i. Similarly, Q is *bounded from above (resp. below, left, right)* by A iff its top (resp. bottom, left, right) edge coincides with the top (resp. bottom, left, right) edge of A. For example, the RR shown dotted in Figure 1 is bounded from above by A, from the left by p_3, from below by p_2 and from the right by p_5.

For reasons which will become clear later, we choose to use two distinct sets of coordinate axes x_1,y_1 and x_2,y_2 as shown in Figure 1. Note that the origin of x_1,y_1 is at the southwest corner of A, while that of x_2,y_2 is at its southeast corner. We use $X_1(p_i), Y_1(p_i)$ to denote the coordinates of p_i with respect to x_1,y_1, while $X_2(p_i), Y_2(p_i)$ denote its coordinates with respect to x_2,y_2. Note that $Y_1(p_i)=Y_2(p_i)$, and that $X_2(p_i)=L-X_1(p_i)$ where L is the horizontal dimension of A (see Figure 1).

A point p_i is said to *dominate* p_j with respect to the coordinates x_1,y_1 iff $X_1(p_i)>X_1(p_j)$ and $Y_1(p_i)>Y_1(p_j)$. We use $DOM_1(p_i)$ to denote the set of points in S that are dominated by point p_i w.r.t. x_1,y_1. $DOM_2(p_i)$ is similarly defined w.r.t. x_2,y_2. In other words, $DOM_1(p_i)$ (resp. $DOM_2(p_i)$) are the points of S that are below and to the left (resp. right) of p_i. Of course

$DOM_1(p_i) \cup DOM_2(p_i)$ is simply the set of points that are below p_i. For example, in Figure 1, $DOM_1(p_3)=\{p_1\}$ and $DOM_2(p_3)=\{p_2,p_4\}$.

If H is a set of points, then a point of H is a *maximum* w.r.t. x_1,y_1 if no other point of H dominates it w.r.t. x_1,y_1. From now on we use $MAX_1(H)$ to denote the set of maxima of H w.r.t. x_1,y_1. $MAX_2(H)$ is similarly defined w.r.t. x_2,y_2. For example, for the set S of points shown in Figure 1, $MAX_1(S)=\{p_4,p_5\}$ and $MAX_2(S)=\{p_1,p_3,p_5\}$. Observe that listing the elements of $MAX_1(H)$ by increasing x_1-components is equivalent to listing them by decreasing y_1-components, so that we can unambiguously talk about 'sorting $MAX_1(H)$' without specifying with respect to which coordinate. A similar remark holds for $MAX_2(H)$.

3. The Algorithm

The algorithm enumerates all the *RR*'s, and chooses the largest-area one. View the *RR*'s as being partitioned into two classes: Those that are bounded from above by p_i's, and those that are bounded from above by A. There are $2n+1$ *RR*'s that are bounded from above by A, and it is not hard to enumerate them in time $O(n \log n)$ (see [NLH] for details). However it is more difficult to enumerate all of the *RR*'s that are bounded from above by p_i's in time $O(s+n \log^2 n)$. From now on, we use SR_i to denote the set of *RR*'s that are bounded from above by p_i, and $BEST(i)$ to denote the largest-area *RR* in SR_i. The algorithm *MER* (below) computes the largest-area *RR*.

Algorithm MER

Step 1: Enumerate the *RR*'s that are bounded from above by A, and let $BEST(0)$ be the largest-area one. As pointed out above, this can be done in time $O(n \log n)$.

Step 2: For $i=1, \ldots, n$ do the following: Compute SR_i and let $BEST(i)$ be the largest-area *RR* in SR_i.

Note: Step 2 is done using algorithm *MER1* which is described later in this Section, and whose

running time is $O(s+n\log^2 n)$.

Step 3: Return the largest-area RR among $BEST(0), \ldots, BEST(n)$. This takes time $O(n)$.

End of Algorithm MER

Correctness of the algorithm follows from the fact that every RR either belongs to one of the SR_i's or is bounded from above by A. The crucial issue is that of implementing Step 2 in time $O(s+n\log^2 n)$. The rest of this Section deals with this problem.

Before describing the algorithm *MER1* for computing the $BEST(i)$'s, we make the following observation.

Observation 3.1 Suppose the elements of $MAX_1(DOM_1(p_i))$ are given in sorted order, and also those of $MAX_2(DOM_2(p_i))$. (In Figure 2, $MAX_1(DOM_1(p_2))=\{p_4,p_3,p_9\}$, $MAX_2(DOM_2(p_2))=\{p_7,p_1\}$, and the rectangles in SR_i are shown dotted). Then it is possible to compute SR_i (and hence $BEST(i)$) in time $O(|SR_i|)$.

Proof: First we observe that

$$|SR_i| = \Theta(\ |MAX_1(DOM_1(p_i))| + |MAX_2(DOM_2(p_i))|\ ).$$

Thus it suffices to show that SR_i can be computed in time $O(|MAX_1(DOM_1(p_i))| + |MAX_2(DOM_2(p_i))|)$. This can indeed be done by simultaneously scanning the elements of $MAX_1(DOM_1(p_i))$ by increasing x_1-coordinate, and those of $MAX_2(DOM_2(p_i))$ by increasing x_2-cordinate. This 'scan' is reminiscent of the way two sorted sequences are merged, and we leave its detailed specification to the reader. □

Before describing algorithm *MER1*, we recall the following result of Overmars and Van Leeuwen: There exists a data structure for dynamically maintaining the maxima of a set of points in the plane w.r.t. a set of coordinate axes x,y, such that insertions and deletions take time $O(\log^2 n)$. Such an *augmented tree structure* (as it is called in [OV]) takes $O(n)$ storage space, and can initially be created in time $O(n\log n)$. At any time, the maxima are available at the root, in sorted order. If the points are stored in the augmented tree structure according to their x-

coordinate, then a *split* operation about any vertical line $x=x_0$ can be implemented in time $O(\log^2 n)$. Such a *split* operation results in two augmented tree structures: One for the points to the left of the vertical line, and one for those to its right. A *concatenate* operation also takes $O(\log^2 n)$ time and has the reverse effect of a *split*.

Now we have all the ingredients for describing algorithm *MER1*.

Algorithm MER1

Input: $p_1, \ldots, p_n, A$

Output: $BEST(1), \ldots, BEST(n)$

Step 1: Create two augmented tree structures $T1$ and $T2$, each of which initially contains all of the p_i's. $T1$ (resp. $T2$) is for the maxima w.r.t. x_1,y_1 (resp. x_2,y_2), and the p_i's are stored in it according to their x_1 (resp. x_2) coordinate. As previously mentioned, both $T1$ and $T2$ can be created in time $O(n \log n)$.

Step 2: Sweep a horizontal line from the top edge of A down to the bottom edge of A, and whenever the line encounters a p_i do the following (i)-(v):

(i) Delete p_i from both $T1$ and $T2$. This takes $O(\log^2 n)$ time.

(ii) Split $T1$ about $X_1(p_i)$, resulting in $T1\_left$ and $T1\_right$. Note that the elements of $MAX_1(DOM_1(p_i))$ are available in sorted order at the root of $T1\_left$ (note that this is true only because all the points above p_i have already been encountered by the downward-sweeping line and hence deleted). This takes time $O(\log^2 n)$.

(iii) Similarly to (ii), split $T2$ about $X_2(p_i)$ and obtain $T2\_left$ and $T2\_right$. Note that $MAX_2(DOM_2(p_i))$ are available in sorted order at the root of $T2\_right$. Again, this takes time $O(\log^2 n)$.

(iv) Obtain SR_i and $BEST(i)$ from $MAX_1(DOM_1(p_i))$ and $MAX_2(DOM_2(p_i))$, in time $O(|SR_i|)$ (Observation 3.1).

(v) Reconstruct $T1$ by concatenating $T1\_left$ and $T1\_right$. Similarly reconstruct $T2$ from $T2\_left$ and $T2\_right$. This takes time $O(\log^2 n)$.

End of Algorithm MER1

The cost of *MER1* is dominated by that of Step 2, whose total time is

$$O(\sum_{i=1}^{n}(|SR_i|+\log^2 n)=O(s+n\log^2 n).$$

This implies that algorithm *MER* itself takes $O(s+n\log^2 n)$ time.

4. Conclusion

We have given an $O(s+n\log^2 n)$ time algorithm for solving the maximum empty rectangle problem, where s is the number of RR's. Our algorithm can be combined with the algorithms of Naamad et. al. and Chazelle et. al. to give an $O(\min(s+n\log^2 n, s\log n, n\log^3 n))$ time algorithm for this problem.

References

[CDL] B. M. Chazelle, R. L. Drysdale and D. T. Lee, 'Computing the Largest Empty Rectangle,' to appear in *SIAM Journal on Computing*.

[NLH] A. Naamad, D. T. Lee and W.-L. Hsu, 'On the Maximum Empty Rectangle Problem,' *Discrete Applied Mathematics*, 1984, pp. 267-277.

[OV] M. H. Overmars and J. Van Leeuwen, 'Maintenance of Configurations in the Plane,' *Journal of Computer and System Sciences*, 1981, pp. 166-204.

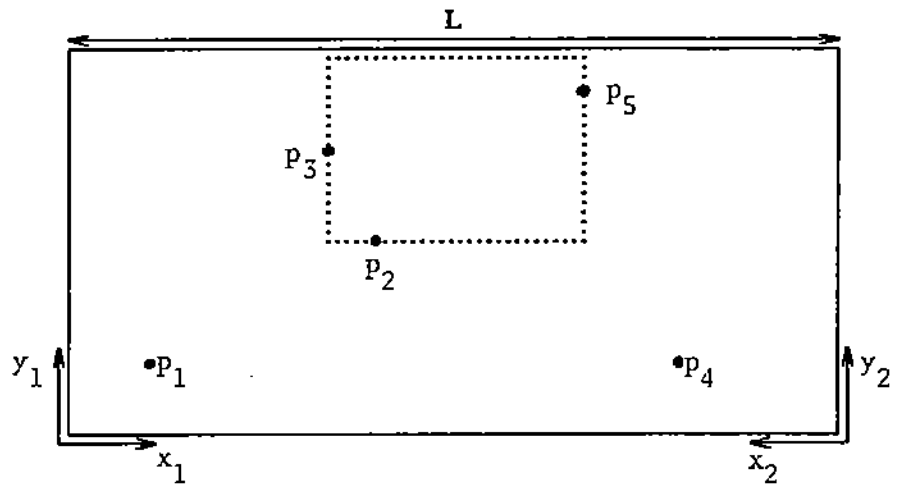

Figure 1

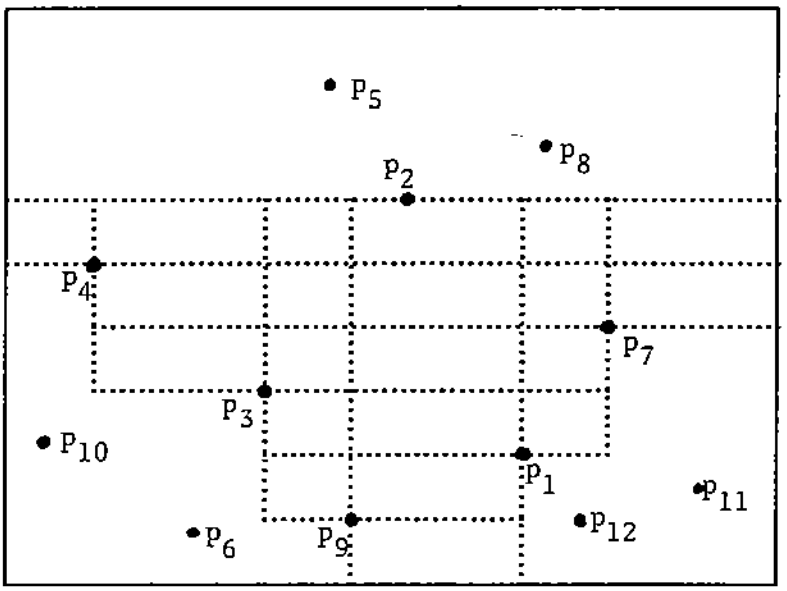

Figure 2