Computing the maximal elements in a point set

Michiel Smid[*]

January 18, 2021

1 Introduction

Let S be a set of n points in the plane. Each point p of S is given by its x- and y-coordinates p_x and p_y, respectively.

A point p of S is called *maximal* in S if there is no point in S that is to the north-east of p, i.e.,

$$\{q \in S \setminus \{p\} : q_x \geq p_x \text{ and } q_y \geq p_y\} = \emptyset.$$

See Figure 1 for an example. Observe that, in general, there is more than one maximal element in S.

We will give two algorithms that compute all maximal elements in S. The first one uses the *divide-and-conquer* paradigm, whereas the second one computes the maximal elements *incrementally*.

2 A divide-and-conquer algorithm

A (very) high-level description of this algorithm is given in Figure 2.

Before we can completely specify the algorithm, we have to answer the following questions:

- How to compute the line ℓ?
- What happens during the merge step?
- How to represent the output of the algorithm?

First, we consider the third question. Let us say that the algorithm returns the maximal elements of the set S in a list, sorted from left to right.

Observation 1 *The order of the maximal elements by x-coordinates is the same as the reversed order of the maximal elements by y-coordinates.*

---

[*] School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6. E-mail: `michiel@scs.carleton.ca`.

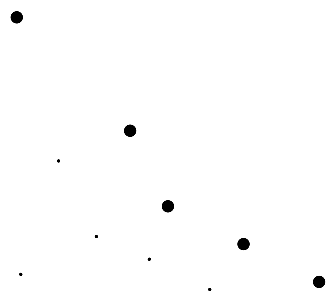

Figure 1: *The •-points are maximal; the ·-points are not maximal.*

```
Algorithm $max(S, n)$
if $n = 1$
then return the only element of $S$
else compute a vertical line $\ell$ such that both
    $A = \{p \in S : p$ left of $\ell\}$ and $B = \{p \in S : p$ right of $\ell\}$
    contain $n/2$ elements;
    $max(A, n/2)$;
    $max(B, n/2)$;
    merge step
endif
```

Figure 2: *The basic structure of a divide-and-conquer algorithm that computes the maximal elements.*

It is easy to answer the first question: At the start, we sort all points of S from left to right, and store them in an array $S[1 \dots n]$. Then the line ℓ is the vertical line through the point $S[n/2]$.

Let us now look at the merge step. Consider the point sets A and B, and assume that we have already computed the maximal elements in A and the maximal elements in B. See Figure 3.

Observation 2 *Each maximal element in B is also maximal in S.*

Observation 3 *Let q be the maximal element in B with the smallest x-coordinate, and let p be an arbitrary maximal element in A. Then p is a maximal element in S if and only if $p_y > q_y$.*

Observation 4 *Each maximal element in S is either maximal in A or maximal in B.*

Based on these three observations, it is easy to write down the merge step. The complete divide-and-conquer algorithm is given in Figure 4. The call $max(S, 1, n)$ computes the maximal elements of the entire point set S.

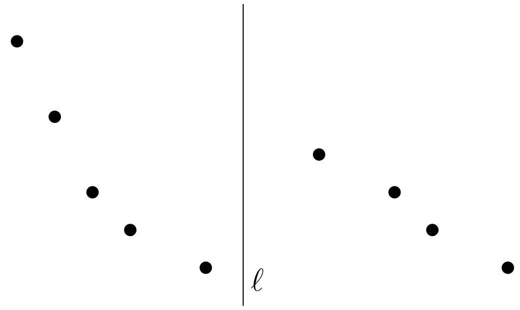

Figure 3: *The points to the left of ℓ are maximal in A; the points to the right of ℓ are maximal in B.*

```
Algorithm $max(S, i, j)$
(* $S[1 \ldots n]$ contains points, sorted from left to right;
   $1 \leq i \leq j \leq n$; the algorithm returns a list containing the
   maximal elements in the set $S[i \ldots j]$, sorted
   from left to right *)
if $i = j$
then return list containing the point $S[i]$
else $k = \lfloor (i + j)/2 \rfloor$;
     $L_1 = max(S, i, k)$;
     $L_2 = max(S, k + 1, j)$;
     $p$ = first point in the list $L_1$;
     $q$ = first point in the list $L_2$;
     if $p_y \leq q_y$
     then return list $L_2$
     else while $p_y > q_y$
            do $p$ = successor($p, L_1$)
            endwhile;
            $p$ = predecessor($p, L_1$);
            return the part of $L_1$ from the beginning to $p$, followed by $L_2$
     endif
endif
```

Figure 4: *The divide-and-conquer algorithm that computes the maximal elements in a point set.*

Remark 1 There may be points having the same x-coordinate. Therefore, we should sort the points *lexicographically.*

Exercise 1 In the while-loop, there is the assignment

$$p = \text{successor}(p, L_1).$$

What should you do if p is the last element p in L_1?

We now analyze the running time of the algorithm. Sorting the points lexicographically takes $O(n \log n)$ time.

Let $T(n)$ be the worst-case running time of algorithm $max(S, i, j)$ for a *sorted* input set of size $j - i + 1 = n$. Then there are constants c and c' such that

$$T(n) \leq \begin{cases} c & \text{if } n = 1, \\ c'n + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) & \text{if } n \geq 2. \end{cases}$$

To solve this recurrence, we assume for simplicity that n is a power of two, i.e., $n = 2^k$. Furthermore, we assume that $c = c' = 1$. If n is large, then

$$\begin{aligned} T(n) &\leq n + 2\,T(n/2) \\ &\leq n + 2\,(n/2 + 2\,T(n/4)) \\ &= 2n + 2^2\,T(n/2^2) \\ &\leq 2n + 2^2\left(n/2^2 + 2\,T(n/2^3)\right) \\ &= 3n + 2^3\,T(n/2^3) \\ &\leq 4n + 2^4\,T(n/2^4) \\ &\leq 5n + 2^5\,T(n/2^5) \\ &\vdots \\ &\leq kn + 2^k\,T(n/2^k) \\ &= n \log n + n\,T(1) \\ &= n \log n + n \\ &\leq 2n \log n \\ &= O(n \log n). \end{aligned}$$

Hence, given a sorted input set, the algorithm takes $T(n) = O(n \log n)$ time. Since the initial sorting step takes $O(n \log n)$ time (and this is done right at the beginning), we have shown that the complete time to compute the maximal elements in a set of n points is $O(n \log n)$.

3 An incremental algorithm

Let us first make the following assumption.

Assumption 1 All points in S have distinct x-coordinates and distinct y-coordinates. That is, no two points are on a horizontal or vertical line.

Here is a simple (but slow) algorithm that computes the maximal elements in S. Consider a point p in S. We can test for all points $q \in S \setminus \{p\}$, if q is to the north-east of p. If this is not the case for all these points q, then p is a maximal element in S. Otherwise, p is not a maximal element. If we repeat this for all points p in S, then we have found all maximal elements in S. The running time of this algorithm is $O(n^2)$. Why?

Observation 5 *If point q is to the left of point p, then q is not to the north-east of p, i.e.,*

$$q_x < p_x \implies q \textit{ not to the north-east of } p.$$

If we want to decide if p is maximal, then it suffices to test for each point q that is to the right of p, whether q is to the north-east of p. (That is, we do not have to consider points q that are to the left of p.) This observation still leads to a quadratic algorithm. Why?

Observation 6 *Let $p \in S$ and let q be the point with maximum y-coordinate that is to the right of p. Then*

$$p \textit{ is maximal in } S \iff p_y > q_y.$$

Proof. Assume that p is maximal in S. Then there is no point $r \in S \setminus \{p\}$ with $r_x > p_x$ and $r_y > p_y$. (Here, we use Assumption 1.) We know that $q_x > p_x$. It follows that $q_y < p_y$.

Conversely, assume that $p_y > q_y$. Since q is the highest point to the right of p, the following holds: For each point $r \in S$ with $r_x > p_x$, we have $r_y \leq q_y$. Hence, for each point $r \in S$ with $r_x > p_x$, we have $r_y < p_y$. This means that there is no point $r \in S \setminus \{p\}$ such that $r_x > p_x$ and $r_y > p_y$. Hence, p is maximal in S. ∎

What do we know? We can use the highest point q that is to the right of p, to decide whether or not p is a maximal element. In fact, if we know q, then we can decide in *one* comparison whether or not p is a maximal element. Of course, this leads to the following question: How do we find for each point p in S the corresponding point q? Here is the answer: We take care that during the algorithm the following invariant holds:

Invariant: If the algorithm considers the point p, then

- all maximal elements to the right of p have already been found, and
- q is the highest point in S that is to the right of p.

The algorithm is given in Figure 5. The correctness follows from the discussion above. What about the running time? We can sort the points in $O(n \log n)$ time. Each iteration of the for-loop takes $O(1)$ time. Since there are $n - 1$ iterations, the entire for-loop takes $O(n)$ time. Hence, the total running time is

$$O(n \log n) + O(n) = O(n \log n).$$

```
**Algorithm** $max(A, n)$
(* The array $A[1 \ldots n]$ contains a sequence of $n$ points *)
sort the points from left to right, and store
the sorted sequence in $A$;
output the point $A[n]$;     (* $A[n]$ is a maximal element *)
$q = A[n]$;
for $i = n - 1$ downto 1
do (* test, if point $A[i]$ is maximal *)
    if $A[i]_y > q_y$
    then output $A[i]$;     (* $A[i]$ is a maximal element *)
            $q = A[i]$
    endif
endfor
```

Figure 5: *This algorithm computes the maximal elements in a point set.*

Remark 2 It is important that we visit the points *from right to left.* Why?

Remark 3 We have assumed that all points in S have distinct x-coordinates and distinct y-coordinates. In general, we sort the points *lexicographically*, i.e.,

$$p \leq_\ell q \iff (p_x < q_x) \text{ or } (p_x = q_x \text{ and } p_y < q_y).$$

Using this ordering, the algorithm does not change.