Lecture Notes: 2D Maxima

Yufei Tao
Department of Computer Science and Engineering
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

Jan 8, 2014

In this lecture, we will discuss the *maxima problem* defined as follows. Let (x_1, y_1) and (x_2, y_2) be two different points in $\mathbb{R}^2$. We say that the former *dominates* the latter if $x_1 \geq x_2$ and $y_1 \geq y_2$ (note that the two equalities cannot hold simultaneously because these are two different points). Let P be a set of n points in $\mathbb{R}^2$. A point $p \in P$ is a *maximal point* of P if p is not dominated by any point in P. We want to design an algorithm to report all the maximal points of P efficiently. In the example of Figure 1, points 1, 2, 6, and 8 should be reported. We will assume that P is in general position (in particular, no two points have the same x-coordinate, or the same y-coordinate). It is not hard to observe that the maximal points must form a "staircase", namely, if we walk along them in ascending order of x-coordinate, their y-coordinates must be descending.

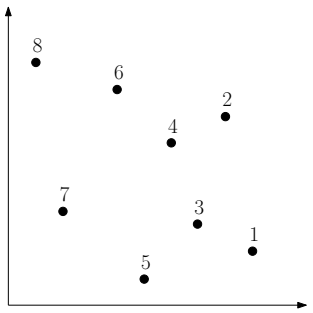

Figure 1: An example

Whether a point $p \in P$ is dominated by any other point in P can be easily checked in $O(n)$ time. This implies a naive algorithm that finds all the maximal points in $O(n^2)$ time. We can, in fact, settle the problem in $O(n \log n)$ time. First sort P in descending order of x-coordinate. Let the sorted order be $p_1, p_2, ..., p_n$. Then, we process the points in the sorted order by adhering to the invariant that, after having processed p_i, we have computed all the maximal points of $\{p_1, ..., p_i\}$, and stored them in descending order of x-coordinate. For example, in Figure 1, after finishing with point 4, we should be maintaining a sorted list: point 1, point 2. This turns out to be very easy. First, p_1 must be a maximal point. In general, suppose that the invariant holds after p_i, we process p_{i+1} as follows. Let p^* be the highest point of $p_1, ..., p_i$ (p^* can be easily maintained). Observe that p_{i+1} is a maximal point *if and only if* the y-coordinate of p_{i+1} is greater than that of p^*. Hence, we can decide whether to append p_{i+1} to our maximal list in just $O(1)$ time. The total time spent on processing $p_1, ..., p_n$ is thus $O(n)$. The overall running time is $O(n \log n)$, which is determined by sorting.

It can also be shown that any comparison-based algorithm must use $\Omega(n \log n)$ time solving the problem. Hence, the algorithm explained earlier is already optimal. Although it may seem that we have solved the problem, our lecture has actually just started. We will give an *output sensitive* algorithm that finishes in $O(n \log k)$ time, where k is the number of maximal points. Attention

should be paid to the methodology behind this algorithm, which is very useful in designing output sensitive algorithms. Before continuing, let us first observe a simple $O(nk)$ time algorithm. First, find the rightmost point p of P in $O(n)$ time, which must be a maximal point. Then, in $O(n)$ time remove from P all the points dominated by p and also p itself. Now, the rightmost point of (the remaining) P is also guaranteed to be a maximal point. Hence, we repeat the above steps until P becomes empty.

1 Utilizing An Upper Bound of k

Let us first assume that, by magic, we know an upper bound $\hat{k}$ of k (e.g., $\hat{k} = n$ is a trivial upper bound). We will design an algorithm whose efficiency depends on $\hat{k}$.

First, divide P by x-coordinate into $\hat{k}$ subsets $P_1, ..., P_{\hat{k}}$ such that (i) every point in P_i has a larger x-coordinate than all the points in P_j for any $1 \leq i < j \leq \hat{k}$, and (ii) $|P_1| = |P_2| = ... = |P_{\hat{k}-1}| = \lceil n/\hat{k} \rceil$ (note that this condition implies $P_i = O(n/\hat{k})$ for all $i \in [1, \hat{k}]$). This can be easily done in $O(n \log \hat{k})$ time using a standard rank selection algorithm (see appendix).

Next, we process the subsets P_i in ascending order of i. Our invariant is that, after we are done with P_i, we must have computed the maximal points of $P_1 \cup ... \cup P_i$ (observe that they must also be maximal points of P). We achieve the purpose as follows. First, all the maximal points of P_1 are found in $O(t_1|P_1|) = O(t_1 n/\hat{k})$ time, where t_1 is the number of those points. In general, assuming that the invariant holds after P_i, we process P_{i+1} as follows. Let p^* be the highest of all the maximal points in $P_1 \cup ... \cup P_i$. Scan P_{i+1} to remove all the points dominated by p^*. Then, find all the maximal points of the *remaining* P_{i+1} in $O(t_{i+1}|P_{i+1}|) = O(t_{i+1} n/\hat{k})$ time, where t_{i+1} is the number of those points—note that all these points must also be maximal points of $P_1 \cup ... \cup P_{i+1}$. Overall, we spend $O((n/\hat{k}) \sum_{i=1}^{k} t_i) = O((n/\hat{k}) \cdot k) = O(n)$ time.

We thus have proved:

Lemma 1. *If an upper bound $\hat{k}$ of k is known, we can find all the maximal points in at most $cn \log \hat{k}$ time for some constant c.*

You may be puzzled why the lemma states the constant c explicitly—usually, we hide such constants with big-O. There is, however, a good reason to do so, as will be clear in the next section.

2 The Final Algorithm

Lemma 1 is not immediately helpful—after all, if we set $\hat{k}$ to the trivial bound n, then the running time $O(n \log \hat{k})$ is no better than $O(n \log n)$, which we have already achieved. However, a more clever use of the lemma leads to the desired $O(n \log k)$ bound. The main idea is to ask the algorithm take a guess k' of k. Initially, the algorithm sets k' to 1, and gradually increases it if $k' < k$. But how expensive is it to find out whether $k' < k$? The answer is $O(n \log \hat{k})$, thanks to Lemma 1. Specifically, we simply run the algorithm of Section 1 by setting $\hat{k} = k'$, and keep monitoring the algorithm's cost (this means counting the number of unit-time atomic operations in the RAM model). We know if $k' \geq k$, then by Lemma 1, the algorithm should terminate within $cn \log \hat{k}$ time. Hence, as soon as the algorithm's cost reaches $1 + cn \log k'$, we can manually force the algorithm to terminate, and declare that $k' < k$.

Motivated by this, we start with $k' = 2^1$. If $k' < k$, we increase k' to 2^2 and try again. In general, if $k' = 2^{2^i}$ is still smaller than k, the next k' we will try is $\min\{2^{2^{i+1}}, n\}$. Clearly, this

algorithm will eventually find all the maximal points—it does so when k' is at least k for the first time.

At first glance, it may appear that the running time is expensive because we may have to attempt multiple values of k' before the algorithm stops. A careful calculation reveals that this intuition is not true. Suppose that eventually the algorithm stops at $k' = 2^{2^i}$. The total running time is:

$$\begin{aligned} & O\left(n \log 2^{2^0} + n \log 2^{2^1} + n \log 2^{2^2} + n \log 2^{2^3} + ... + n \log 2^{2^i}\right) \\ = \; & O\left(n\left(2^0 + 2^1 + 2^2 + ... + 2^i\right)\right) \\ = \; & O(n \cdot 2^i) \end{aligned}$$

How large is 2^i? The definition of i implies $2^{2^{i-1}} < k$, namely, $2^{i-1} < \log_2 k$. Hence, $2^i < 2 \log_2 k$. We thus have designed an algorithm solving the maxima problem in $O(n \cdot 2^i) = O(n \log k)$ time.

Appendix: Multi-Rank Selection

Let S be a set of n real values. We say that a value $v \in S$ has rank i if $|\{u \in S \mid u \geq v\}| = i$ (i.e., the largest value in S has rank 1, the second largest rank 2, ...). Given any rank $r \in [1, n]$, the element with rank r can be selected in linear time $O(n)$ using a textbook rank selection algorithm (note that this does *not* require sorting S).

In the *multi-rank selection problem*, suppose we are given k ranks $r_1, ..., r_k$ in ascending order, and need to find the k corresponding elements. This is do-able in $O(n \log k)$ time as follows. Without loss of generality, let us assume that k is a power of 2. We first pick the median of $r_{k/2}$ of $\{r_1, ..., r_k\}$, and find the element e with rank $r_{k/2}$. Then, divide S into S_1 and S_2 such that (i) the former includes all the elements of S at least e, and (ii) the latter includes the other elements of S. We now recurse on two instances of the multi-rank selection problem: the first one on S_1 with ranks $r_1, ..., r_{k/2}$, and the second one on S_2 with ranks $r_{1+k/2} - k/2, r_{2+k/2} - k/2, ..., r_k - k/2$.

Let us analyze the running time. Define $f(n, k)$ be the time of the above algorithm issued on a set S ($n = |S|$), and k designated ranks. If $k = 1$, we know $f(n, k) = O(n)$. For $k > 1$, we have:

$$f(n, k) \;\; = \;\; f(n_1, k/2) + f(n - n_1, k/2)$$

where $n_1 = |S_1|$. Solving the recurrence gives $f(n, k) = O(n \log k)$.