CS264: Beyond Worst-Case Analysis
Lecture #2: Instance-Optimal Geometric Algorithms*

Tim Roughgarden†

September 24, 2014

1 Preamble

This lectures touches on results of Afshani, Barbay, and Chan [1], who give a number of interesting instance-optimality results for fundamental problems in computational geometry, namely the problems of computing the maximal points or the convex hull of a point set in two or three dimensions. These are perhaps the most compelling examples to date of instance-optimal algorithms when the cost measure $\text{cost}(A, z)$ is the running time of an algorithm. We discuss only their simplest result, for the following *2D Maxima* problem; this suffices to showcase most of the paper's main ideas.

2 The Problem and the Goal

The input is n points in the plane. Assume for simplicity that all coordinate values are distinct (this is not important). Say that x is *dominated* by y if y is bigger in both coordinates (i.e., lies to the northeast of x). A *maximal point* is one not dominated by any other.[1] The goal is to compute the set of all maxima of the point set. See Figure 1.

We use the comparison model, familiar from the study of sorting algorithms. That is, we assume that an algorithm can access the input only through comparisons of coordinate values, and define $\text{cost}(A, z)$ as the number of comparisons that the algorithm A needs to compute the correct answer for the input z.[2][3]

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

[1]Students of economics know these as "Pareto optimal" points.

[2]OK, so we lied slightly in Section 1 when we said that cost would measure the running time.

[3]The results in [1] for the convex hull problem use, for both the upper and lower bounds, a somewhat richer set of primitives.

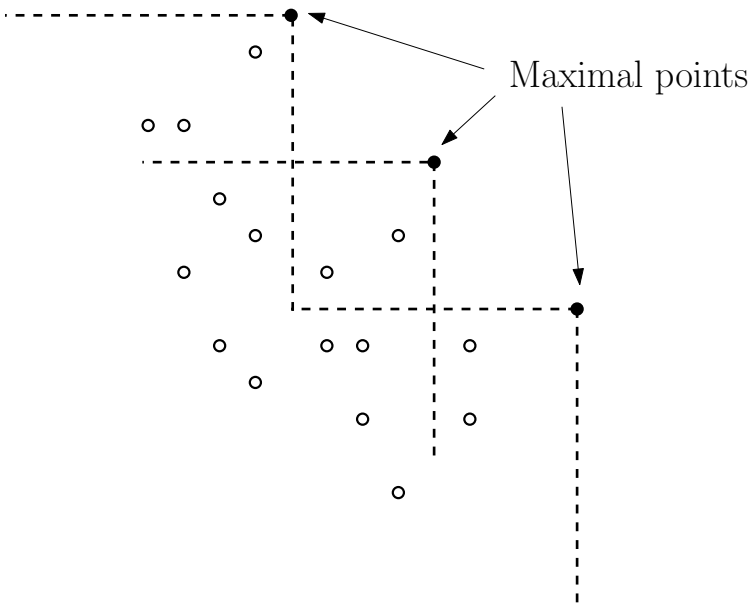

Figure 1: A point set and its maxima. Solid circles are the maximal points, hollow circles are dominated points. The dashed lines indicate the "region of domination" for each of the maximal points.

Our goal is to prove that there is an instance-optimal algorithm A for the 2D Maxima problem. Recall what this means from last lecture: there is a constant $c \geq 1$ such that, for every "natural" algorithm B,

$$\text{cost}(A, z) \leq c \cdot \text{cost}(B, z).$$

Proving that an algorithm is instance optimal involves two tasks: an upper bound for an algorithm A on every input z, and a matching (up to a constant factor) lower bound on every natural algorithm B and every input z. A prerequisite for a matching lower bound is an input-by-input tight upper bound on the performance $\text{cost}(A,)$ of algorithm A (why?). This is a much stronger running time guarantee than the worst-case guarantees that you're probably accustomed to, and most of this lecture is devoted to such an upper bound. We'll only have time for a few comments on the lower bound, and defer the interested reader to the paper [1] for details.

3 The KS Algorithm and Its Correctness

One neat aspect of the main result of this lecture is that it proves the instance optimality of an existing algorithm for the problem — the classic and clever algorithm of Kirkpatrick and Seidel [2], which we review next.

We claim that the KS algorithm, as described in Figure 2, correctly solves the 2D Maxima problem. First, note that all deletions are of dominated points and are therefore justified.

Input: a point set Q.

1. If $Q = \emptyset$ return; if $|Q| = 1$ return Q.
2. Split the input into left and right halves Q_ℓ and Q_r, by computing the median x-coordinate among points of Q.
3. Let q have the maximum y-coordinate in Q_r, and add q to the output set S.
4. Delete q and everything it dominates.
5. Recurse on (what's left of) Q_ℓ and Q_r.

Figure 2: The Kirkpatrick-Seidel (KS) algorithm.

Second, observe that the point q is maximal in the point set Q: its x-coordinate is bigger than everything in Q_ℓ, and its y-coordinate is bigger than everything in Q_r. Thus, we only need to argue that neither deletions nor recursing transforms a point from non-maximal to maximal. Recursively computed maxima of Q_r are also maxima in Q: such a point p is not dominated by anything in Q_r, nor by q (since it wasn't pruned), nor by anything in Q_l (all of which have smaller x-coordinates than p). Finally, recursively computed maxima of Q_ℓ (post-pruning) are also maxima in Q: such a point p is not dominated by anything in Q_l, nor by q (since p was not deleted), nor by anything in Q_r (since q has larger y-coordinate than any points in Q_r). Thus every point is eventually either correctly deleted or correctly identified as a maximal point of the original point set.

4 Running Time Analysis of the KS Algorithm

An $O(n \log n)$ Bound. First, observe that the work done — in particular, the number of comparisons made — in a call to the KS algorithm (finding a median, a maximum, and pruning) is linear in the input size, not counting the work done by further recursive calls. Each recursive call is on at most half of the input. The standard MergeSort recurrence $T(n) \leq 2T(n/2) + O(n)$ gives an upper bound of $O(n \log n)$ on the running time of the KS algorithm.

An $O(n \log h)$ Bound. In Homework #1 you will show that the running time of the KS algorithm is $\Omega(n \log n)$ in the worst case. But some inputs are easy: for the point set in Figure 3, there is only one maximal point, which is identified in the outermost recursive call. On this input, the KS algorithm terminates in $O(n)$ time. We conclude that some points sets are easier than others — at the least for the KS algorithm, if not more fundamentally.

Recall that a prerequisite for an instance-optimality result is a tight input-by-input upper bound on the performance of the allegedly optimal algorithm. The $O(n \log n)$ upper bound,

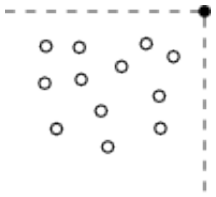

Figure 3: An easy point set where the KS algorithm runs in $O(n)$ time.

while best-possible among bounds that are parameterized solely by the input size n, is not going to cut it. We need a more tightly parameterized upper bound.

The next idea is to parameterize the running time of the KS algorithm both by the input size n *and by the output size* — the number of maxima — denoted by h. These are sometimes called *output-sensitive* time bounds.

We claim that the KS algorithm runs in $O(n \log h)$ time on every input. The informal proof is as follows. Every recursive call successfully identifies a new maximal point (the point q). Thus the jth level of recursion identifies 2^j new maximal points. Thus there can only be $\log h$ recursion levels, and the total work at each level is $O(n)$.

Why is this not a proof? Because it is only recursive calls with *non-empty input* that identify new maximal points. The pruning step might render one (or both) recursive calls vacuous, which can cause the number of recursion levels to exceed $\log k$. We leave a formal proof of the $O(n \log h)$ bound to Homework #1.

A Still More Refined Bound. We know that the $O(n \log h)$ running time bound is tight in the worst case, but it is not tight input-by-input (see below and Homework #1). To prove an instance-optimality result, we need an even more precisely parameterized running time bound on the KS algorithm. To state the parameterization, suppose that we can partition the input set S into groups $S_1, \ldots, S_k$ where for each i:

1. either S_i is a single point; or

2. there is an axis-aligned box B_i such that its interior contains all of S_i and lies strictly below the "staircase" of S (recall Figure 1).[4]

We call such a partition *legal*; see Figure 4 for an example. The intuition is that each group S_i of the second type represents a cluster of points that can conceivably be eliminated by an algorithm such as the KS algorithm in one fell swoop.[5] Thus the bigger the S_i's, the easier one might expect the instance to be. Formally, we prove the following.

[4]Don't forget that this partitioning is for analysis purposes only — the algorithm remains the five lines of code in Figure 2.

[5]Beginning from the northeast corner of the box, travel north until you hit the staircase, and then east until you hit a maximal point q. The point q dominates all of the points in S_i.

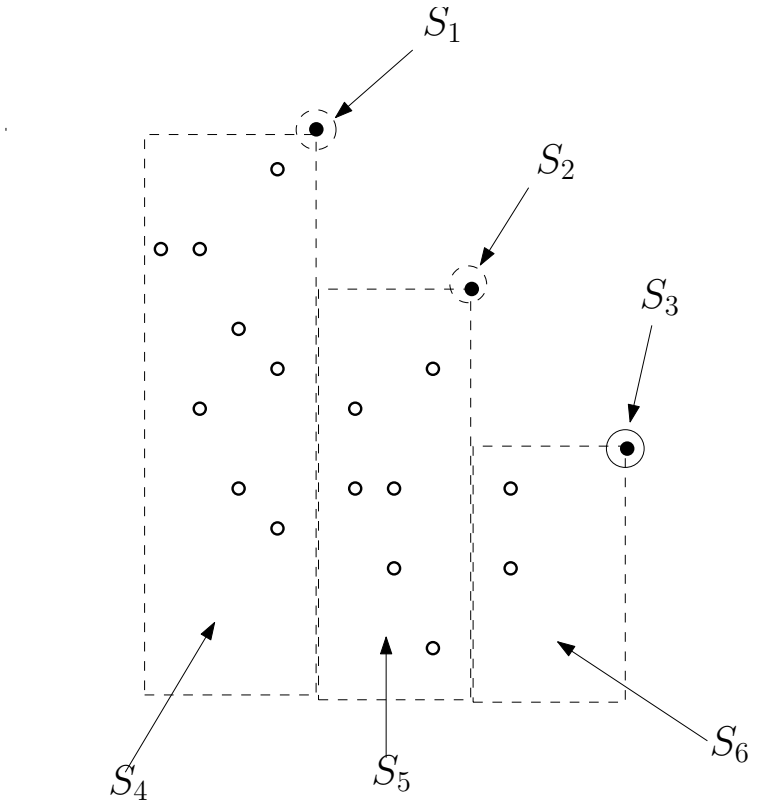

Figure 4: A legal partition of a point set.

Theorem 4.1 ([1]) *For every point set S, the running time of the KS algorithm is*

$$O\left(\min_{\text{legal } \{S_i\}} \sum_{i=1}^{k} |S_i| \log \frac{n}{|S_i|}\right). \tag{1}$$

Theorem 4.1 is a very fine-grained upper bound on the running time of the KS algorithm, parameterized by the "easiness" of the input point set. To get a feel for the expression in (1), note that it is certainly $O(n \log n)$ for every n-point set S — just partition the point set into singletons. It is also $O(n \log h)$ for every n-point set S with h maxima: take the S_i's to be "vertical slabs" as in Figure 4 and use the convexity of the function $x \log x$ (see Homework #1). It is tempting to interpret the expression in (1) as some type of "entropy measure" for point sets, albeit a measure specific to the problem of computing maxima.

Proof of Theorem 4.1: Consider a legal partition, as above. The proof amounts to a very clever method of accounting for all the work done over the course of the KS algorithm. We bound each set S_i's contribution to the running time separately, and then sum over all of the S_i's.

Consider a set S_i. The set's contribution to the work done at recursion level j is linear in the number of points of S_i that have not yet been pruned by the algorithm. Obviously there

are at most $|S_i|$ such points. The key step in the proof is the following alternative upper bound.

Key Claim: Every set S_i has $O(n/2^j)$ unpruned points remaining at the jth recursion level of the KS algorithm.

Let's prove the theorem assuming the key claim. Observe that the bound of $|S_i|$ is better than the bound of $n/2^j$ for the first $\approx \log_2(n/|S_i|)$ recursion levels. Summing over all recursion levels j, the total contribution of the points of S_i to the running time of the KS algorithm is

$$\leq \underbrace{|S_i| + |S_i| + \cdots + |S_i|}_{\log(n/|S_i|) \text{ times}} + \frac{|S_i|}{2} + \frac{|S_i|}{4} + \frac{|S_i|}{8} + \cdots$$

which is $O(|S_i| \log \frac{n}{|S_i|})$. Summing over all the S_i's proves the theorem.

Proof of the Key Claim: Fix the recursion level j. We break the proof into two bite-size pieces.

(1) All of S_i's remaining points are sandwiched (in x-coordinate) between two adjacent (in x-coordinate) maxima-so-far.

(2) For every pair of adjacent (in x-coordinate) maxima-so far, there are $O(n/2^j)$ remaining points sandwiched between them.

To prove (1), let a, b denote the x-coordinates of the maximal points identified so far that flank the right-hand side of S_i's box B_i (see Figure 5). Take $a = -\infty$ or $b = +\infty$ if q_1 or q_2 doesn't exist. By definition, all points of S_i have x-coordinate less than b. Since the interior of B_i lies below the staircase of S (recall Figure 4) and q_1 is a maximal point, q_1 lies at or above the top of B_i. Points of S_i with x-coordinate less than a were pruned when q_1 was identified (if not earlier), so all surviving points of S_i have x-coordinate bigger than a.

To prove (2), induction on j shows that the $\leq 2^j$ level-j recursive calls bucket by x-coordinate the surviving points into $\leq 2^j$ non-empty buckets with population $\leq n/2^j$ each (Figure 6). Each of these recursive calls identifies a maximal point in its bucket. After these maxima are identified, there are at most $2 \cdot n/2^j$ surviving points strictly between two successive maximal points (which occurs with two consecutive buckets that have maximal points at the "far left" and "far right", respectively).

This completes the proof of (2), and hence of the Key Claim, and hence of Theorem 4.1. ■

5 Instance Optimality of the KS Algorithm

Theorem 4.1 gives a fine-grained parameterized upper bound on the running time of the KS algorithm. Proving that the KS algorithm is instance optimal in the comparison model is

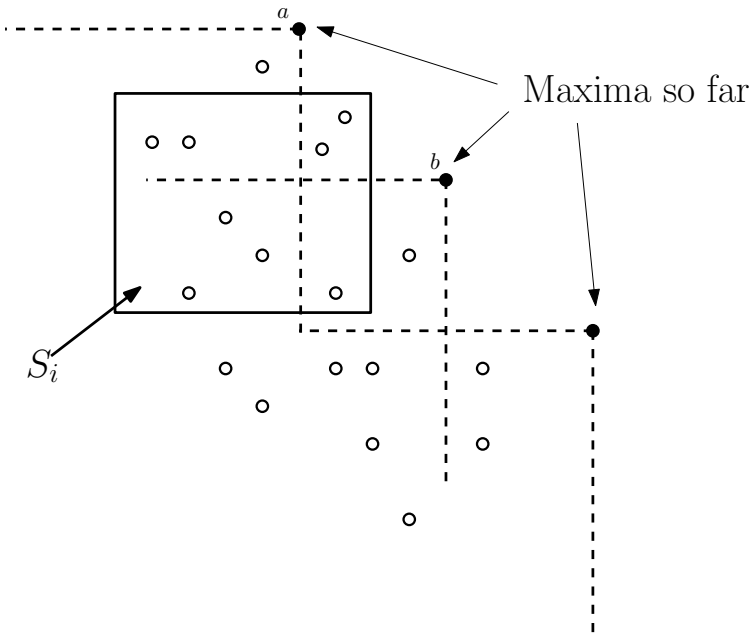

Figure 5: Proof of key claim (part 1) for Theorem 4.1.

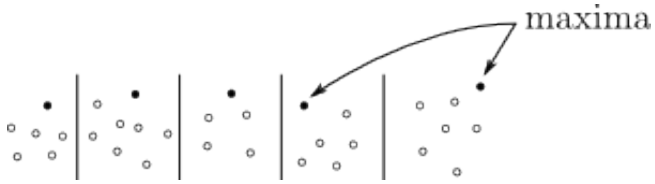

Figure 6: Proof of key claim (part 2) for Theorem 4.1, showing 5 of the 2^j buckets formed by the level-j recursive calls.

equivalent to proving that, for every input z, every (correct) algorithm B requires

$$\Omega\left(\min_{\text{legal } \{S_i\}}\left\{\sum_i |S_i| \log \frac{n}{|S_i|}\right\}\right) \tag{2}$$

comparisons to compute the maximal points of the point set specified by z.

But there is a problem: *this is clearly false*. The irritating reason is that an algorithm can "memorize" (i.e., have hard-coded within) the correct answer for a particular input z, and regurgitate it without any non-trivial work. Precisely, for a fixed input z, consider the following algorithm (which is defined for every input w):

1. Check if the real input w is z.

2. If so, output the (hard-coded) correct answer for z.

3. If not, run (say) the KS algorithm on w.

This algorithm is clearly correct. Since each of the first two steps takes linear time, this algorithm runs in linear time on the input z. We see that the ambition of instance optimality is so strong that lower bounds arise for trivial reasons.[6]

A general pep talk: *annoying counterexamples are not a good reason to abandon the quest for an interesting theorem.* If you had a particular theorem in mind, perhaps the model or the theorem statement can be modified slightly to obtain a true and meaningful result? Afshani et al. [1] persevered in the face of the stupid example above and proved an instance-optimality-like guarantee for the KS algorithm.

The motivation for the guarantee formulated and proved in [1] is the following observation: our running time bound for the KS algorithm depends only on the *set* S of input points, and is independent of the *order* in which these points happened to listed in the input to the algorithm. On the other hand, the stupid algorithm above does *not* have this property: if only the unordered set S were memorized in advance, then the first step (verifying that the input is in fact S, listed in some order) would require $\Omega(n \log n)$ comparisons (why?) and the algorithm would pose no barrier to an instance optimality result. Intuitively, one expects the running time bound of a "natural" algorithm to be "order-oblivious" in the same sense as our analysis of the KS algorithm.

Now, an obvious (but still ambitious) goal would be to formulate a definition of a "natural" or "order-oblivious" algorithm, and then show the following:

Theorem 5.1 (Informal Version [1]) *For every input z of* 2D Maxima*, every "order-oblivious" algorithm requires*

$$\Omega\left(\min_{\text{legal } \{S_i\}} \left\{ \sum_i |S_i| \log \frac{n}{|S_i|} \right\}\right)$$

comparisons.

This is precisely the spirit of the results of Afshani et al. [1], and thus *the KS algorithm is instance optimal among the class of order-oblivious algorithms.*

The precise statement in [1] is even cooler. There remains the lingering question of how to formally define an "order-oblivious" algorithm. This issue is sidestepped in [1] by proving a stronger result. Rather than restricting the class of possible algorithms, we allow an *arbitrary* algorithm B and evaluate its running time under a *worst-case ordering* of a given point set S. In our abstract notation, we now think of $\text{cost}(B, \cdot)$ as a vector indexed not by (ordered) inputs but by (unordered) point sets, and we define

$$\text{cost}(B, S) = \max_{\pi} \{\text{cost}(B, \pi(S))\}, \tag{3}$$

[6] Why does this problem not occur with the Threshold Algorithm (Homework #1)? Because in that setting we are only interested in sublinear time bounds anyway — by the time an algorithm verified the input (as in Step 1 above), the threshold algorithm likely would have stopped early and left it in the dust.

where S denotes a point set, $\pi(S)$ denotes the input in which the points of S are specified by the ordering π, and $\text{cost}(B, \pi(S))$ denotes the number of comparisons used by B to correctly solve the input $\pi(S)$. Thus we (conceptually) run the algorithm $|S|!$ different times — once for each ordering of S — and take the largest number of comparisons used. Obviously, for every "order-oblivious" algorithm — one for which (our upper bound on) its running time is order-independent, like the KS algorithm — this extra max operator has no effect. But an algorithm that memorizes an input no longer performs well by this measure. Indeed, the KS algorithm is "order-oblivious instance-optimal" in the following sense.

Theorem 5.2 (Formal Version [1]) *For every point set S and every (correct) algorithm B,*

$$\textit{cost}(B, S) = \Omega\left(\min_{\textit{legal } \{S_i\}} \left\{\sum_i |S_i| \log \frac{n}{|S_i|}\right\}\right)$$

where $\textit{cost}(B, S)$ is defined as in (3)*.*

In fact, Afshani et al. [1] prove an even stronger instance-optimality result for the KS algorithm, which replaces the right-hand side of (3) by the average of $\text{cost}(A, \pi(S))$ over all orderings π. Thus, for every point set, no algorithm can beat the lower bound of (2) on a "typical" ordering.

There are two approaches to proving Theorem 5.2, both of which can be made to work. The first method is based on the following intuition: every correct algorithm must implicitly justify every input point that it does not output, by identifying some other point that dominates it. Thus the transcript of every correct algorithm encodes a proof of non-maximality of all the non-maximal points. Recalling the intuition behind the definition of legal partitions, it is plausible that the expression in (2) is related to the length of such proofs, and it can be shown that the proof length necessary (for a worst-case ordering) is indeed bounded below by (2). The second proof approach uses an adversary argument. Here, we consider an arbitrary point set S and the best legal partition $\{S_i\}$ for it, and an arbitrary algorithm B. We simulate B, and adversarially resolve comparisons to adaptively hide the maxima. That is, as long as the number of comparisons used by B so far is too small (less than (2)), we can order the points of S so that B cannot yet be sure of what the correct answer is, because there remain two inputs consistent with the comparisons so far but with different sets of maximal points. The details are non-trivial but readable; see [1].

6 Conclusions

6.1 Key Contributions

The 2D Maxima analysis in [1] has three main points.

1. A novel and fine-grained measure of the "input complexity" of a point set — with respect to the 2D Maxima problem — and a point set-by-point set matching upper

bound for the running time of the KS algorithm. We can think of this measure (2) as a "condition number" for the 2D Maxima problem; we'll return to this idea several times in future lectures.

2. A model that sidesteps the annoying issue of "algorithms that memorize the correct answer." An interpretation of the solution in [1] is that only "order-oblivious" algorithms are allowed as competitors, but the actual statement is stronger and more elegant.

3. A proof that, point set-by-point set, the above measure of input complexity lower bounds the number of comparisons of every algorithm under a worst-case (or even average-case) ordering of the point set.

We emphasize that it's pretty amazing that there are instance-optimal algorithms — with any reasonable version of the definition — for such fundamental problems. This seems to be the exception, rather than the rule.

7 Discussion and Take-Home Points

1. Instance optimality is an input-by-input guarantee, and as such is essentially the strongest notion of optimality one could hope for, and the only uncontroversial notion of optimality. (Of course, the value of the suppressed constant factor can also be important.)

2. Because of its strength, few instance optimality guarantees are known. Finding more, across all sorts of problem domains and cost measures, is a great research direction.

3. The first reason that instance-optimal results appear rare is that they simply don't exist in many cases (see Homework #1). This accords with practice where, for many computational problems, the "right" algorithm really does seem to depend on the domain (i.e., on the set of inputs).

   The definition of instance optimality can be weakened in various ways to make such guarantees more feasible. Section 5 considered point set-by-point set guarantees, rather than input-by-input guarantees. More generally, one can have groups of inputs, and ask about group-by-group guarantees. The definition of "natural algorithms" also provides a knob to turn, interpolating between bigger classes of algorithms (and hence stronger instance-optimality results) and smaller ones (where instance-optimality results are more likely to exist).

4. The second reason that instance-optimal algorithms seem rare is that, even when they do exist, proving instance-optimality might be out of reach of all known proof techniques. Recall that a prerequisite for an instance optimality result is an input-by-input lower bound on the performance of every algorithm. Thus, one should only seek instance optimality results in models where there are lower bound techniques. For running time analyses, this would seem to limit the possibilities to problems solvable in

close to linear time (as in this lecture); even here, we worked in the comparison model so that super-linear lower bounds were readily available. In computational models that admit information-theoretic lower bounds — like in Problem #1, or in online algorithms, streaming algorithms, and mechanism design — there would seem to be many more opportunities.

References

[1] P. Afshani, J. Barbay, and T. M. Chan. Instance-optimal geometric algorithms. In *Proceedings of the 50th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 129–138, 2009.

[2] D. G. Kirkpatrick and R. Seidel. Output-size sensitive algorithms for finding maximal vectors. In *Proceedings of the First Annual ACM Symposium on Computational Geometry*, pages 89–96, 1985.