E0 208: Computational Geometry

Naman Mishra

August 2024

Contents

Lectures

The course

Course webpage

MS Teams: 9ng333s

Lecture 1.
Wednesday
August 7

Grading

Tentatively

(30%) 4 assignments

(30%) 1 midterm

(40%) Final / project

Overview

Applications to

- Robotics
- VLSI
- Databases (spatial)
- Machine learning

Examples.

- **Robotics:** Path planning
- **Databases:** Given a set of restaurants with ratings, find the top k restaurants within a distance r from a given location.
  - Processing a single query takes $O(n)$,
  - and preprocessing leads to space issues.

  It seems we are fed.

- **Nearest-neighbour query:** Given a set of n points and a query point q, return the point closest to q. (Voronoi diagram)

The first half focuses on these data structure based problems. The second half focuses on geometric optimization queries.

Examples.

- **Travelling salesman** in the Euclidean setting.
- **Set cover:** Given a set of points and a set of disks, find the smallest set of disks that covers all points.
- **Convex hull/Skyline points/Arrangement of lines**

Conferences

- **SoCG:** Symposium on Computational Geometry
- **CCCG:** Canadian Conference on Computational Geometry

Chapter I

Skyline points

Definition I.1 (Domination and skyline)**.** A point $p = (p_1, p_2, \ldots, p_d)$ *dominates* another point $q = (q_1, q_2, \ldots, q_d)$ if $p_i > q_i$ for all $i \in [d]$.

A point p in a set S is a *skyline point* if no other point in S dominates it.

Question I.2. *Given a set of points in $\mathbb{R}^2$, find the set of skyline points.*

Solution. Sort the points by decreasing x-coordinate. Iterate through the list keeping track of the maximum y-coordinate seen so far (sweep left). If the current point has a y-coordinate less than the maximum, it is a skyline point.

```
Skyline(P):
  sort P by decreasing x-coordinate
  S ← ∅
  y_max ← −∞
  for p ∈ P
     if p_2 > y_max
        S ← S ∪ {p}
        y_max ← p_2
```

■

Question I.3. *Given a set of points in $\mathbb{R}^3$, find the set of skyline points.*

Solution. Sort P by decreasing z-coordinate. Whether a point is a skyline point depends only on points before it. Sweep through P, while maintaining the solution set S, and the 2D skyline for the projection of each point in S onto the xy-plane.

This 2D skyline is to be stored in a (balanced) binary search tree according to the x-coordinate. Each time a new point p_i is seen, we query the tree for the successor a_j of p_i. Then p_i is a skyline point iff $(p_i)_y > (a_j)_y$.

If p_i is a skyline point, insert it into the tree, and delete all points dominated by p_i. $O((1+k)\log n)$ time for deleting k points.

Time complexity is

$$\begin{aligned}\sum_{i=1}^{n} O(\log n) + k_i \cdot O(\log n) &= O\left(\sum_{i=1}^{n} \log n\right) + \left(\sum_{i=1}^{n} k_i\right) O(\log n) \\ &= O(n\log n + n\log n) = O(n\log n)\end{aligned}$$

```
3DSkyline(P):
    sort P by decreasing z-coordinate
    S ← ∅, T ← ∅
    for p ∈ P
        p⁺ ← next(T, p)
        if (p⁺)_y < p_y
```

```
create(P):
    huh
```

■

If the number of skyline points is much smaller than the number of points, we can do better than $O(n\log n)$. **Lecture 2.** Monday August 12

In this lecture, we will figure out an $O(n\log k)$ algorithm, where k is the number of skyline points in the input.

Warm-up: If $k = 1$, is there a better algorithm than $O(n\log n)$?
Answer: Yes. Simply pick the maximum x-coordinate.

I.1 A slow algorithm

Question I.4. *Can we find an $O(nk)$ algorithm for the skyline problem? (assuming d to be constant)*

Solution. Pick the largest x-coordinate, and remove all points that are dominated by it. Repeat until the set is exhausted.

The next point chosen is not dominated by the previous one, hence it is not dominated by any other deleted point. Since it has the largest x-coordinate of the remaining points, it cannot be dominated by those either. ■

We will try to make this faster later.

I.2 Chan's algorithm

Question I.5. *Can we find an $O(n \log k)$ algorithm for the skyline problem in $\mathbb{R}^2$?*

We will first solve the following decision problem:

Question I.6. *Given a set of points P having k skyline points and an integer $\widehat{k}$, decide*

- *if $\widehat{k} \geq k$, output the skyline,*
- *if $\widehat{k} < k$, output* ***failure****.*

Solution. Partition P into roughly $\widehat{k}$ slabs of roughly equal size using the median of medians algorithm.

(1) Compute the median and partition the elements in $O(n)$ time.

(2) Repeat recursively on the two halves, upto a depth of $\log k$.

In total, this takes $O(n \log k)$ time. (One remarked that it's like quicksort, but stopping once we have exhausted our time budget.)

Visit each slab in order of decreasing x-coordinate.

```
Chan-Decision-2D(P, k̂):
    P_1, ..., P_k̂ ← partition(P)
    S ← ∅
    y(p*) ← −∞
    for i = k̂ downto 1
        if |S| > k̂
            return Failure
        else
            P'_i ← {p ∈ P_i | y(p) > y(p*)}
            S(P'_i) ← Skyline(P'_i)        ▷ using the slow algorithm
            report S(P'_i)
            S ← S ∪ S(P'_i)
    return S
```

Failure is not because of our inability to output the correct answer, but because the runtime would be too large. ■

Analysis. Constructing the slabs took $O(n \log \widehat{k})$ time.

Running the slow algorithm on P_i takes $\frac{n}{\widehat{k}} \cdot k_i$ time, where k_i is the number of skyline points in P_i. In total, this step takes

$$\sum_{i=1}^{\widehat{k}} O\left(\frac{n}{\widehat{k}} \cdot k_i\right) = O\left(\frac{n}{\widehat{k}} \cdot \sum_{i=1}^{\widehat{k}} k_i\right) = O(n)$$

Lecture 2: Skyline in $O(n \log k)$ (Chan's algorithm) time. Thus the total runtime is $O(n \log \widehat{k})$. ■

We are now prepared to solve the original problem.

Solution. Guess $\widehat{k} = 1, 2, 4, \ldots, 2^{\lceil \log k \rceil}$. This will take

$$\sum_{i=0}^{\lceil \log k \rceil} O(n \log 2^i) = \sum_{i=0}^{\lceil \log k \rceil} O(ni) = O(n(\log k)^2)$$

time. We can be even more aggressive and guess $\widehat{k}_i = 2^{2^i}$ (square the previous guess). This takes

$$\sum_{i=0}^{\lceil \log \log k \rceil} O(n \log 2^{2^i}) = \sum_{i=0}^{\lceil \log \log k \rceil} O(n 2^i) = O\big(n 2^{\lceil \log \log k \rceil}\big) = O(n \log k)$$

time.

```
Chan-2D(P):
  k̂ ← 2
  for ever
      S ← Chan-Decision-2D(P, k̂)
      if S ≠ Failure
          return S
      square k̂
```

■

Question I.7. *Can we generalize this to $\mathbb{R}^3$?*

It suffices to generalize the decision problem.

Solution. We construct $\widehat{k}$ slabs based on the x-coordinate. The invariants are the following:

- S_i will store the skyline of the first i slabs.
- S_i^{yz} will store the 2D skyline of the projection of S_i onto the yz-plane.
- Each S_i and S_i^{yz} is sorted according to the y-coordinate.

The deletion step now takes $|P_i| \log |S_i^{yz}|$ time, (as opposed to $|P_i|$ earlier) which is still fine.

We now run the slow algorithm and report the elements added to S_i. This takes $O(n)$ time, and they are naturally reported in decreasing order of the y-coordinates.

Chan-Decision(P, $\widehat{k}$):

$P_1, \dots, P_{\widehat{k}} \leftarrow$ partition(P)
$S \leftarrow \varnothing$, $S^{yz} \leftarrow \varnothing$
$y(p^*) \leftarrow -\infty$
for $i = \widehat{k}$ downto 1
  if $|S| > \widehat{k}$
    return Failure
  else
    $P'_i \leftarrow$ filter(P_i, S^{yz})
    $S(P'_i) \leftarrow$ Skyline(P'_i) ▷ using the slow algorithm
    report $S_(P'_i)$
    $S \leftarrow S \cup S(P'_i)$
return S

■

Chapter II

Orthogonal range searching

Question II.1. *Given a set of points in $\mathbb{R}^2$ and a rectangle $[x_1, x_2] \times [y_1, y_2]$, report all points in the rectangle.*

Lecture 3. Wednesday August 14

Without preprocessing, we have

- $O(n)$ space,
- $O(n)$ time.

In the worst case, we can have $O(n)$ points in the rectangle, so we cannot do better than $O(n)$ time.

Let k be the number of points in the rectangle. Usually, $k \ll n$. Thus we will design a data structure that allows us to report the k points in $O(f(n)+k)$ time, where $f(n)$ is as small as possible.

II.1 The 1-dimensional case

Question II.2. *Given a set of points in $\mathbb{R}$ and an interval $[x_1, x_2]$, report all points in the interval.*

Solution. Sort the points in $O(n \log n)$ time using only $O(n)$ space. For each query, binary search for x_1 and keep going until x_2. This takes $O(\log n + k)$ time. ■

II.2 The 2-dimensional unbounded case

Question II.3. *Given a set of points in $\mathbb{R}^2$ and an unbounded rectangle $[x_1, x_2] \times [y_1, \infty)$, report all points in the rectangle.*

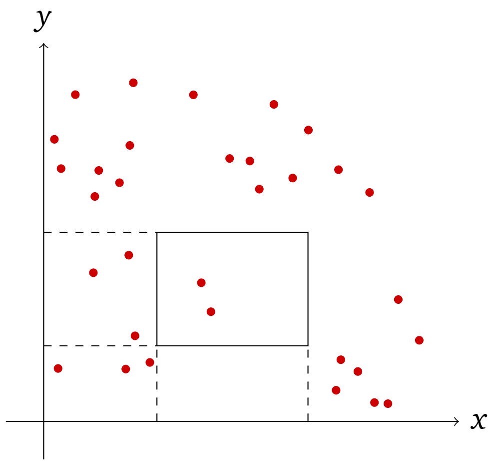

Figure II.1: Orthogonal range searching in $\mathbb{R}^2$.

Naive. Project onto the x-axis and solve the 1-dimensional case. Discard all points with y-coordinate less than y_1.

This takes $O(n \log n)$ preprocessing time. For each query, it takes $O(\log n + \#[x_1, x_2])$ time, where $\#[x_1, x_2]$ is the number of points with x-coordinate in $[x_1, x_2]$. ■

This can be much larger than k, in fact, as large as n.

Solution. Create a *priority search tree* (PST) in $O(n \log n)$ time using $O(n)$ space, and sorted by x-coordinate with y-coordinate as priority.

Each query can be answered as follows.

- Locate the predecessor of x_1 and successor of x_2. To ensure they exist, add sentinels $-\infty$ and ∞ during preprocessing.

- Query each subtree in between with a DFS, searching down until the y-coordinate is less than y_1. ■

Analysis. Querying the subtree $\mathcal{T}$ takes $O(1 + k_{\mathcal{T}})$ time, where $k_{\mathcal{T}}$ is the number of rectangled points in the subtree. This is because each rectangled point is visited exactly once, and any non-rectangled point that is checked is preceded by a rectangled point.

Thus the total time is

$$\begin{aligned} \sum_{\mathcal{T}} O(1 + k_{\mathcal{T}}) &= O\left(\sum_{\mathcal{T}} 1\right) + O(k) \\ &= O(\log n + k). \end{aligned}$$

This is because the each subtree arises from a point on the search path for the predecessor or successor, which has length at most $\log n$.

```
PST-Unbound(P):
    𝒫 ← PST(P ∪ {(−∞, ⊥), (∞, ⊥)})
    u ← root(𝒫), v ← root(𝒫)
    while u = v
        TODO
```

■

II.2.1 Priority search trees

A priority search tree is a binary search tree where each node has an associated priority.

Midterm 1: September 9th.

Homework 1 will be posted by Wednesday.

Lecture 6. Monday August 26

> **Question II.4.** *Let $L_1, L_2, \ldots, L_t$ be t sorted lists, with $\sum_{i=1}^{t} |L_i| = n$. Given a query q, report the successor of q in each list.*

The naive binary search approach is $O(t \log n)$, but with some preprocessing we can better this to $O(\log n + t)$.

Definition II.5. For any list L, we define $\mathrm{Even}(L)$ to be the list of all even-indexed elements of L.

Solution. For each i we define the list L'_i as follows.

$$\begin{aligned} L'_t &= L_t \\ L'_i &= \mathrm{merged}(L_i, \mathrm{Even}(L'_{i+1})) \end{aligned}$$

For example, if

$$\begin{aligned} L_1 &= \begin{bmatrix} 2 & 3 & 7 & 16 & 19 & 32 & 36 & 37 & 48 & 51 \end{bmatrix} \\ L_2 &= \begin{bmatrix} 15 & 25 & 30 & 35 & 40 & 45 \end{bmatrix} \\ L_3 &= \begin{bmatrix} 5 & 10 & 17 & 20 & 27 & 50 \end{bmatrix} \end{aligned}$$

then

$$\begin{aligned} L'_3 &= \begin{bmatrix} 5 & 10 & 17 & 20 & 27 & 50 \end{bmatrix} \\ L_2 &= \begin{bmatrix} 10 & 15 & 20 & 25 & 27 & 30 & 35 & 40 & 45 \end{bmatrix} \\ L'_1 &= \begin{bmatrix} 2 & 3 & 7 & 15 & 16 & 19 & 25 & 30 & 32 \\ 36 & 37 & 40 & 48 & 51 \end{bmatrix} \end{aligned}$$

Let us formally show that the sum of the new lengths is still $O(n)$.

$$\begin{aligned}
|L'_t| &= |L_t| \\
|L'_{t-1}| &\le |L_{t-1}| + \frac{1}{2}|L_t| \\
|L'_{t-2}| &\le |L_{t-2}| + \frac{1}{2}|L'_{t-1}| \\
&= |L_{t-2}| + \frac{1}{2}|L_{t-1}| + \frac{1}{4}|L_t| \\
|L'_{t-i}| &\le |L_{t-i}| + \frac{1}{2}|L'_{t-i+1}| \\
&= |L_{t-i}| + \frac{1}{2}|L_{t-i+1}| + \frac{1}{4}|L_{t-i+2}| + \cdots + \frac{1}{2^i}|L_t|
\end{aligned}$$

Thus the sum of lengths is

$$\sum_i |L'_i| < 2\sum_i |L_i|$$

Call the elements that have cascaded up ($\mathrm{Even}(L'_{i+1})$) the *cascaders*.

While merging L_i with $\mathrm{Even}(L'_{i+1})$, store a pointer, which we will call the *cascader pointer*, from each element $x \in L'_i$ to the largest element $\le x$ in $\mathrm{Even}(L'_{i+1})$. In our example, we store the pointers

2 3 7 15 16 19 25 30 32 36 37 40 48 51

10 15 20 25 27 30 35 40 45

5 10 17 20 27 50

■

II.3 Back to 2-D range trees

Only one midterm, in the middle of September.

Lecture 7. Wednesday August 28

In the pointer machine model, memory is organised in a graph. Chazelle (1990) showed that in a pointer machine model, any algorithm for range reporting that achieves $O(\log^c n + k)$ time must use $\Omega\left(n\frac{\log n}{\log\log n}\right)$ space.

We will achieve this bound in today's lecture.

Question II.6. *Given a set of points in $\mathbb{R}^2$ and an $x_0 \in \mathbb{R}$, process queries of the form $[x_1, x_2] \times [y_1, y_2]$ where $x_1 < x_0 < x_2$.*

Solution. Partition the set of points P into $P_L = P \cap ([-\infty, x_0) \times \mathbb{R})$ and $P_R = P \cap ([x_0, \infty) \times \mathbb{R})$. Use the 3-sided range reporting algorithm to report the points in $[x_1, \infty) \times [y_1, y_2]$ from P_L and $(-\infty, x_2] \times [y_1, y_2]$ from P_R.

This takes $O(\log n + k)$ time and consumes $O(n)$ space. ■

We can generalize this to work for any query, by using a balanced binary search tree on the x-coordinates of the points.

Solution. Let x_m be the median x-coordinate in P.

Partition P into P_L and P_R as before. Recurse on P_L and P_R. For each node, construct the data structure described above: two priority search trees containing all descendants of the node. The space requirement is given by the recursion

$$S(n) \leq O(n) + 2S(n/2)$$

which has solution $S(n) = O(n \log n)$. This can be thought of as $O(n)$ space per level of the tree. The construction time is given by the recursion

$$T(n) \leq O(n \log n) + 2T(n/2)$$

which has solution $T(n) = O(n \log^2 n)$.

For each query, locate the highest node in the tree that intersects the query rectangle. This takes $O(\log n)$ time. Use the data structure at that node to report the entire rectangle in $O(\log n + k)$ time. ■

The key idea is to use a search tree with larger fanout to reduce the height. A tree with fanout f has height $O(\log_f n)$. For a fanout of $\log n$, this is $O\left(\frac{\log n}{\log \log n}\right)$.

Let us look at the special case question II.6 first.

Question II.7. *Given a set of n points in* $\mathbb{R}^2$ *and* $\ell_1, \ldots, \ell_{F-1} \in \mathbb{R}$, *process queries of the form* $[x_1, x_2] \times [y_1, y_2]$ *where* (x_1, x_2) *contains at least one* ℓ_i.

Solution (naive). Split the points into F slabs $L_1, \ldots, L_F$ by the lines $x = \ell_1, \ldots, \ell_{F-1}$.

For each slab, build a priority search tree. Any query $[x_1, x_2] \times [y_1, y_2]$ will overlap with $L_i, \ldots, L_j$ for some $i < j$. Run the 3-sided range reporting algorithm on each of these slabs. This takes $O(n)$ space and $O(F \log n + k)$. Oops! ■

(diagram) The range is three-sided only in the slabs L_i and L_j. The middle slabs are essentially 1-dimensional queries. Thus we can do better.

Solution. For each slab, build a PST. In addition, maintain a fractional cascading structure on the slabs, where each point is projected onto the y-axis. The total space occupied is still $O(n)$.

Any query $[x_1, x_2] \times [y_1, y_2]$ will overlap with $L_i, \ldots, L_j$ for some $i < j$. The intersection will be 3-sided only with L_i and L_j. Solve these using the PSTs in $O(\log n + k)$ time.

Use fractional cascading to obtain the successor of y_1 in each slab in the middle in $O(\log n + F)$ time. Keep going until y_2, in $O(k)$ time. The total time is $O(\log n + F + k)$. ■

If $F = O(\log n)$, we have $O(\log n + k)$ time using $O(n)$ space.

Finally, we can generalize this to any query.

Solution. Let $F = \lceil \log n \rceil$. Partition P into equal $P_1, \ldots, P_F$ by x-coordinate. Recurse on each P_i, storing F PSTs and a fractional cascade of F slabs at each node.

This consumes $O(n)$ space at each level, for a total of $O\left(\frac{n \log n}{\log \log n}\right)$.

At query time, we have two cases:

(Case 1) The query rectangle intersects at least one slab boundary. Solve the special case in $O(\log n + k)$ time.

(Case 2) The query rectangle is contained within a slab. Recurse into the slab.

Figuring out which case we are in takes $O(\log F)$ time. The tree has height $O\left(\frac{\log n}{\log \log n}\right)$, so getting to case 1 takes at most $O\left(\log F \cdot \frac{\log n}{\log \log n}\right) = O(\log n)$ time. Once we are in case 1, reporting requires $O(\log n + k)$ time. Thus we still have $O(\log n + k)$ time using the lower bound space. ■

To recap, we have solved 2D orthogonal range searching using

Lecture 8. Monday September 2

- **Range trees:** $O(n \log n)$ space and $O(\log^2 n + k)$ time, which we later improved to $O(\log n + k)$ using fractional cascading.
- **The crazy optimal:** $O\left(n \cdot \frac{\log n}{\log \log n}\right)$ space and $O(\log n + k)$ time.

What if we are limited to $O(n)$ space?

Question II.8. *Given a set of points in $\mathbb{R}^2$ and $O(n)$ space, process queries of the form $[x_1, x_2] \times [y_1, y_2]$.*

II.4 Kd-tree

Given a set of points $P \subseteq \mathbb{R}^2$, split them evenly by a vertical cut. Next, split the two halves by a horizontal cut. Repeat this process recursively. Create a tree with the cuts as nodes and the points as leaves.

The construction time is $O(n \log n)$.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The space used is $O(n)$.

$$S(n) = 2S\left(\frac{n}{2}\right) + O(1)$$

We can do this for any number of dimensions.

```
Kd-tree(d, P, a):
  m ← median({p.a | p ∈ P})
  P_ℓ ← {p ∈ P | p.a ≤ m}
  P_r ← {p ∈ P | p.a > m}
  T_ℓ ← Kd-tree(d, P_ℓ, a + 1 mod d)
  T_r ← Kd-tree(d, P_r, a + 1 mod d)
  return node(a, m, T_ℓ, T_r)
```

Solution. Store the points in a Kd-tree. To each internal node v, we have an associated rectangle $R(v)$ containing all points in the subtree rooted at that node.

For a query rectangle $Q = [x_1, x_2] \times [y_1, y_2]$, we traverse the tree from the root, while keeping track of the rectangles $R(v)$.

- At a leaf node, determine if the point is in the rectangle.
- At an internal node v

  (1) If $Q \cap R(v) = \varnothing$, ignore the subtree.

  (2) If $Q \supseteq R(v)$, report all points in the subtree.

  (3) If only part of $R(v)$ intersects Q, recurse on both children. Equivalently, the boundary of Q intersects $R(v)$.

```
Kd-query(v, Q, R):
  if v is a leaf
      report v if v ∈ Q and return
  if Q ∩ R = ∅
      return
  if v splits by x-coordinate
      R_ℓ ← R ∩ [−∞, v.m] × ℝ
      R_r ← R ∩ [v.m, +∞] × ℝ
  else
      R_ℓ ← R ∩ ℝ × [−∞, v.m]
      R_r ← R ∩ ℝ × [v.m, +∞]
  Kd-query(v.ℓ, Q, R_ℓ)
  Kd-query(v.r, Q, R_r)
```

■

Remark. Cases (2) and (3) are identical from the algorithm's perspective. Both involve recursing on both children. They are distinguished here for the purpose of analysis.

Analysis. We take $O(k)$ time across all nodes of type (2).

The parent of every node visited (including leaves) is of type (3), except the subtrees in case (2) already accounted for above. Thus the total number of nodes visited can be bounded by 2 times the number of type (3) nodes. We will bound this number.

How many nodes can intersect a line $x = L$? Fix a node v. If v is split by x-coordinate, at most one child intersects $x = L$. If v is split by y-coordinate, both children may intersect. Thus the number of intersecting nodes at most doubles every 2 levels. This gives a total of $O(2^{\frac{\log n}{2}}) = O(\sqrt{n})$ nodes. More formally,

$$I(n) \leq 3 + 2I\left(\frac{n}{4}\right),$$

which has solution $I(n) = O(\sqrt{n})$. The 3 is for the root and possibly both children, but only two of the grandchildren can intersect.

Thus there are at most $O(\sqrt{n})$ nodes of type (3). This gives total time $O(\sqrt{n} + k)$. ■

In the 3D case, the number of nodes doubles twice every 3 levels, giving $O(n^{2/3})$ nodes intersecting an axis-perpendicular plane, and a time complexity of $O(n^{2/3} + k)$. In d dimensions, we have $O(n^{1-1/d} + k)$ time.

$$I(n) \leq (2^d - 1) + 2^{d-1} I(n/2^d) \implies I(n) = O(n^{\frac{d-1}{d}})$$

This is because whenever a cut is parallel to the place, there is no increase in the number of intersecting nodes.