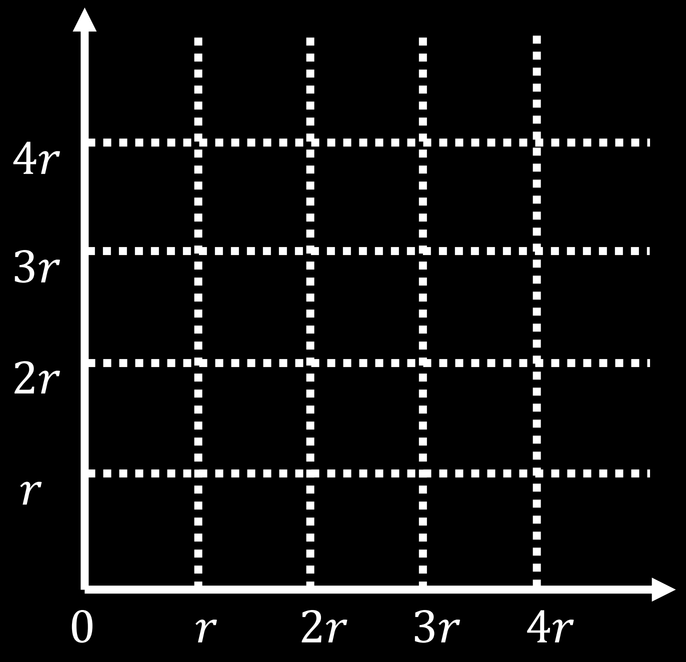
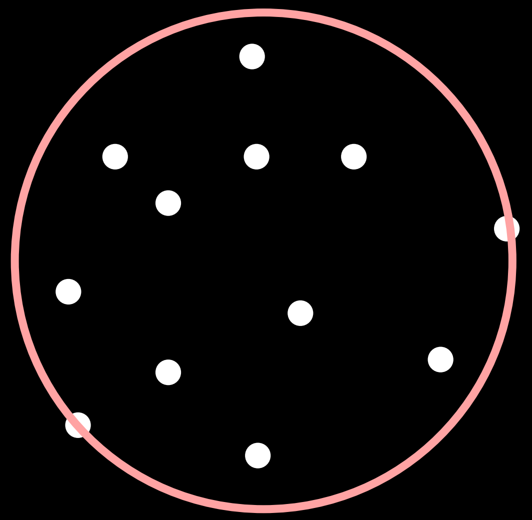
Lecture 22: Computational Geometry II

Randomized incremental algorithms





Goals for today

- Apply **randomized incremental algorithms** to geometry
- Give randomized incremental algorithms for two key problems:
  - The **closest pair** problem
  - The **smallest enclosing circle** problem
- Use **backward analysis** to analyze the runtime of these algorithms

Model and assumptions

- Points are real-valued pairs (x, y)
- Arithmetic on reals is $O(1)$ again
- We can take the floor function of a real in $O(1)$ time
- Hashing is $O(1)$ time in expectation (see universal hashing)

Closest Pair

The closest pair problem

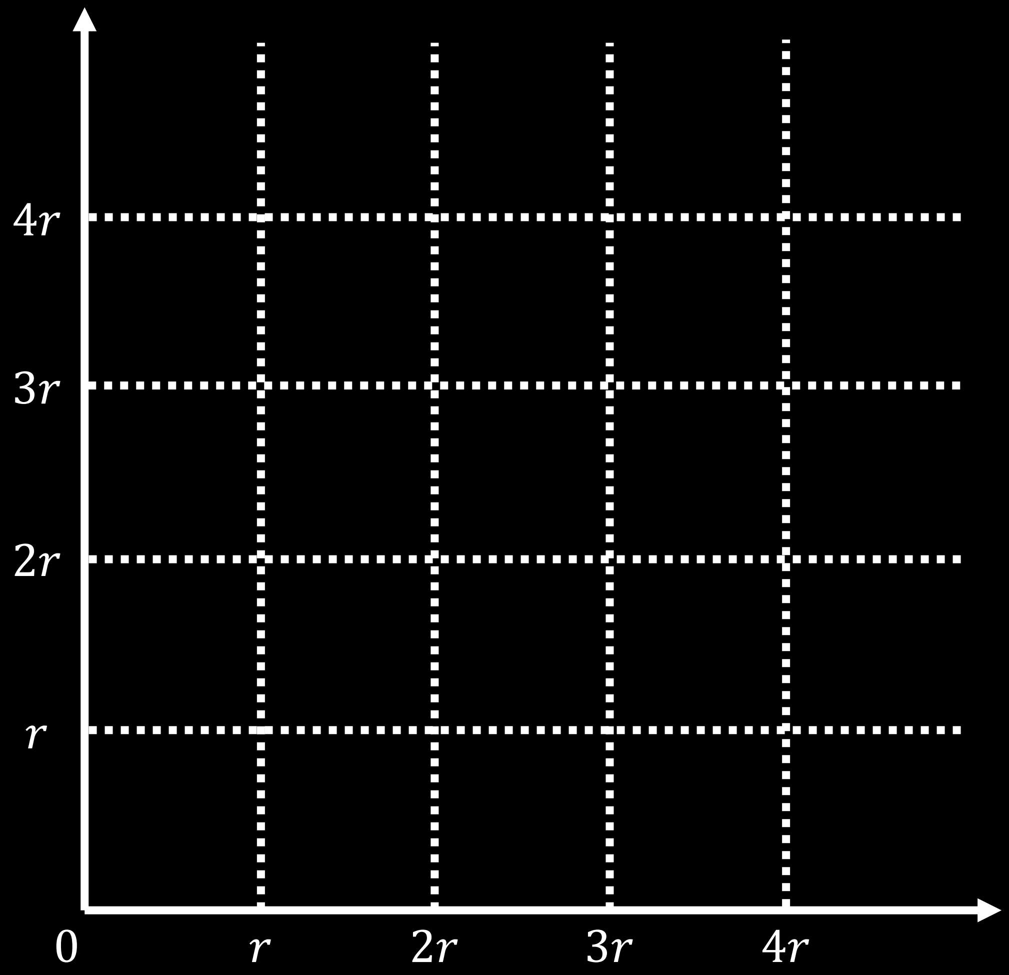
Problem (closest pair): Given n points P, define $CP(P)$ to be the closest distance, i.e.

$$CP(P) = \min_{p,q \in P} ||p - q||$$

Goal is to compute $CP(P)$

A grid data structure

Let's define a grid with size r



How does this help?

- If the grid size is sufficiently large, closest pair will be in same cell, or in neighboring cells
- If the grid size is too large, there will be too many points per cell…

Goal: Choose the right grid size.

- Want few points per cell, so that looking in a cell is fast
- Want the closest pair to be in neighboring cells so we find them fast

The right grid size

Claim (the right grid size): Given a grid with points P and grid size $r = CP(P)$, no cell contains more than four points

Proof:

An incremental approach

Key idea (incremental): Add the points one at a time

- Check neighboring cells to see if there's a new closest pair
- If so, rebuild the grid with the new size
- Otherwise keep going

A grid data structure

Invariant (grid size): Given a grid containing a set of points P, we want the grid size r to always equal $CP(P)$

- MakeGrid(p, q): Make a grid containing p and q, with $r = ||p - q||$
- Lookup(G, p): Given a grid G and point p (not currently in the grid), we want to know whether p is part of a new closest pair
- Insert(G, p): Given a grid G and point p, inserts p and returns the grid size (which may have changed because of p)

Implementing the grid

Issue: The number of grid cells could be unbounded…

Implementing the grid

Implement MakeGrid(p, q):

Implementing the grid

Implement Lookup(p, q):

Implementing the grid

Implement Insert(p, q):

Runtime

Claim (runtime): The worst-case runtime of the incremental grid algorithm is $O(n^2)$

Proof:

Randomization to the rescue!!!

Randomized runtime

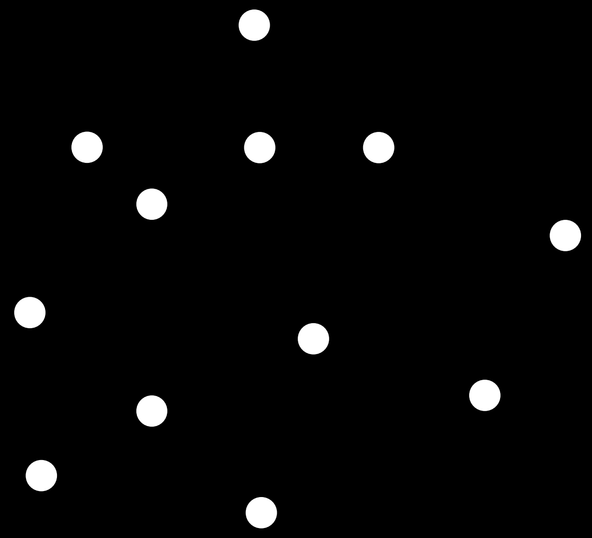
Claim (randomized incremental is fast): If we randomly shuffle the points, then run the incremental grid algorithm, it takes $O(n)$ time in expectation

Proof:

Smallest enclosing circle

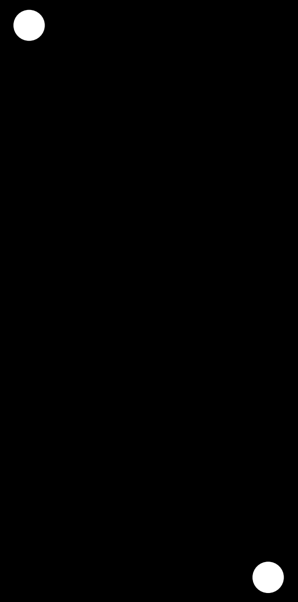
The smallest enclosing circle

Problem (Smallest enclosing circle): Given $n \geq 2$ points in two dimensions, find the smallest circle that contains all of them



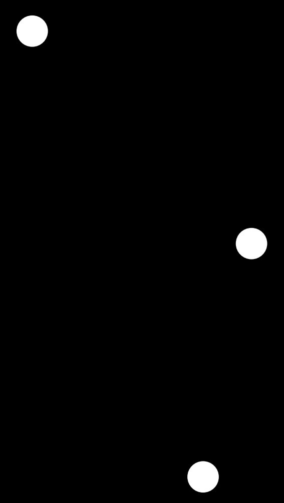
Base cases

Base case (two points):

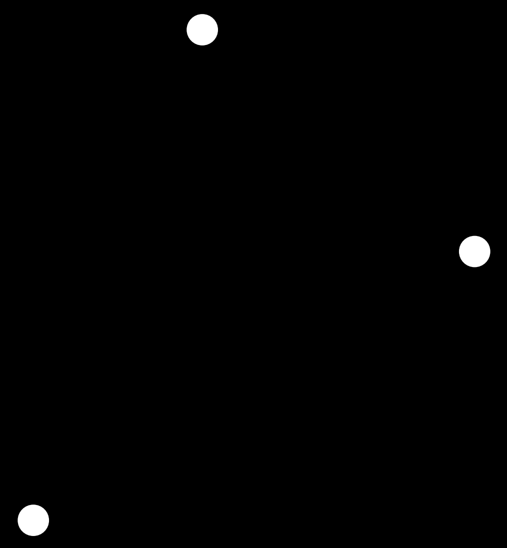


Base cases

Base case (three points):



Case 1: Obtuse angle



Case 2: Acute angle

Three points and a circle

Fact (unique circle): Given three non-colinear points, there is a unique circle that goes through them

The general case

Given $n > 3$ points, how many circles do we need to consider?

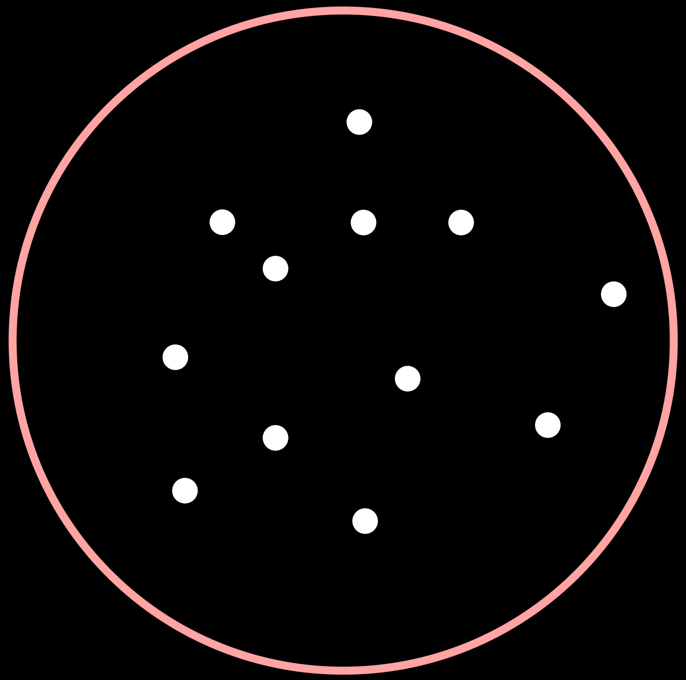
Theorem (three points is always enough): For any set of points, the smallest enclosing circle either touches two points p_i, p_j at a diameter, or touches three points p_i, p_j, p_j

In other words: For any set of points, there exists i, j, k, such that

$$SEC(p_1, \ldots, p_n) = SEC(p_i, p_j, p_k)$$

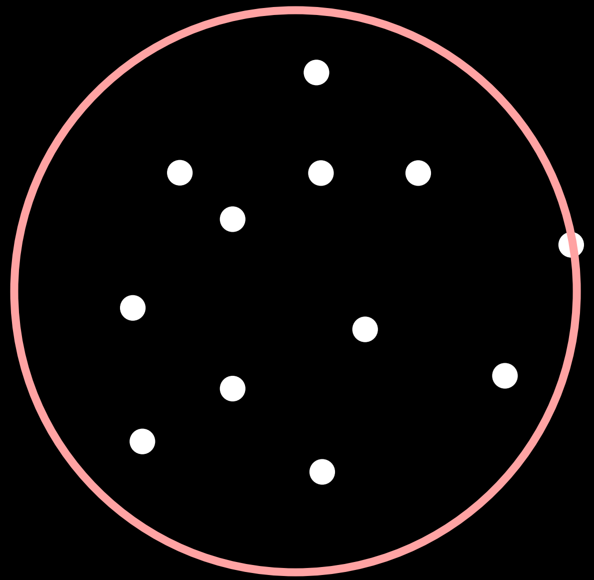
Proof of theorem

Case 1 (no points):



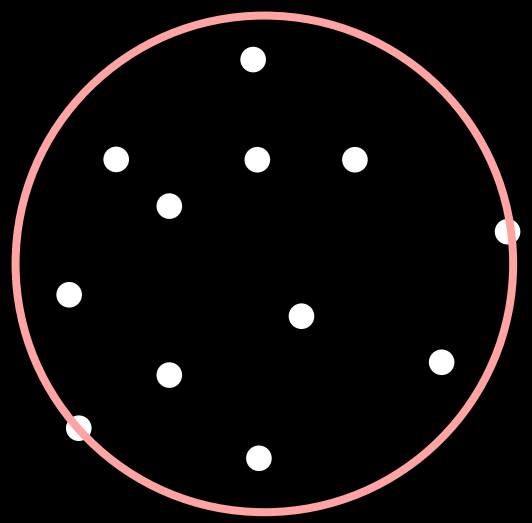
Proof of theorem

Case 2 (one point):



Proof of theorem

Case 3 (two point):



Proof of theorem

Case 4 (three or more points):

Brute force algorithms

Algorithm 1 (brute force): Try all triples of points and find their smallest enclosing circle. Check whether this circle contains every point. Returns the smallest such circle.

Algorithm 2 (better brute force): Try all triples of points and find their smallest enclosing circle. Return the **largest** such circle.

Beating brute force: incremental

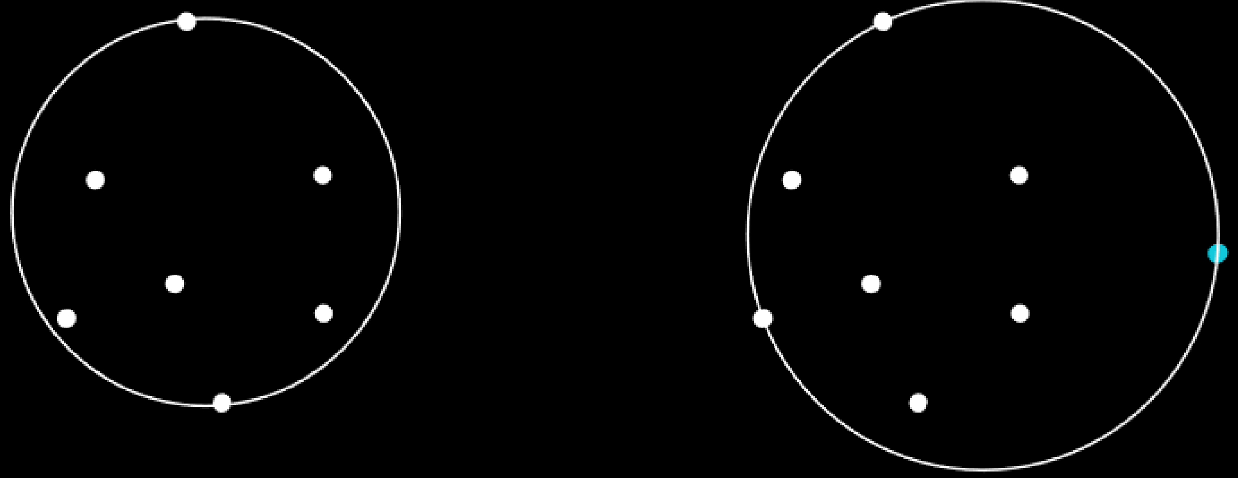
Incremental approach: Insert points one by one and maintain the smallest enclosing circle

When inserting p_i:

- ***Case 1:*** p_i is inside the current circle. Great, do nothing!
- ***Case 2:*** p_i is outside the current circle. Need to find the new one

Making incremental fast

Observation: When we add p_i, if it is not in the current circle, then it is on the boundary of the new circle



Incremental algorithm

SEC($[p_1, p_2, \ldots, p_n]$) = {

Let C be the smallest circle enclosing p_1 and p_2

for *i = 3 to n do* {

if p_i is not inside C **then** $C =$ ______________________

}

return C

}

Incremental algorithm continued

SEC1($[p_1, p_2, \dots, p_k], q$) = {

  Let C be the smallest circle enclosing p_1 and q

  for *i = 2 to k do* {

    if p_i is not inside C **then** $C =$ ________________________

  }

  return C

}

Incremental algorithm deeper again

SEC2($[p_1, p_2, \dots, p_k], q_1, q_2$) = {

Let C be the smallest circle enclosing q_1 and q_2

for *i = 1 to k do* {

if p_i is not inside C **then** $C =$ ____________________

}

return C

}

Runtime

Runtime (SEC2): SEC2 runs in $O(k)$ time

Runtime (SEC1): In the worst case, SEC1 runs in $O(k^2)$ time

Runtime (SEC): In the worst case, SEC runs in $O(n^3)$ time

Randomization to the rescue!!!

Claim (randomized SEC is fast): If we randomly shuffle the points in SEC and SEC1, then SEC1 runs in $O(k)$ expected time and SEC runs in $O(n)$ expected time

Summary

- **Randomized incremental algorithms** are pretty great. We can turn slow brute force algorithms into expected linear-time algorithms!
- We got $O(n)$ time for **closest pair** and **smallest enclosing circle**
- **Backward analysis** helps us analyze the runtime of these randomized incremental algorithms