Randomized Incremental Algorithms

- Incremental Algorithm
  - Process the objects one at a time to solve problem
  - Objects might not be in an order causing bad worst case time complexity

- Randomized Incremental Algorithm
  - Permute the objects randomly
  - Objects not likely to be in a bad order
  - Good average time complexity

Smallest Enclosing Disc
- Given a set of points find the smallest enclosing disc.

Nonincremental Algorithm
- For every two and three points construct the disc through the points and check that all the other points are inside.
- Pick the smallest of these discs.
- \( O(n^4) \) time.

Incremental Algorithm
- If the new point is outside the current disk then it is on the boundary of the new disc.
Incremental Algorithm 0

MinDisc0(p1, p2, ..., pn)
Let D0 be the smallest disc containing p1 and pn
For i = 1 to n do
  if pi in D0 then
    D0 := D0
  else
    D0 := MinDisc1((p1, p2, ..., pi-1, pi))
  Return D0

MinDisc1((p1, p2, ..., pi-1, pi, qi)) returns the smallest disc that contains (p1, p2, ..., pi-1, qi) with pi on the boundary.

Incremental Algorithm 1

MinDisc1(p1, p2, ..., pn, pi)
Let D0 be the smallest disc containing p1 and pn
For i = 2 to n do
  if pi in D0 then
    D0 := D0
  else
    D0 := MinDisc2((p1, p2, ..., pi-1, pi, pi))
  Return D0

MinDisc2((p1, p2, ..., pi-1, pi, qi, pj)) returns the smallest disc that contains (p1, p2, ..., pi-1, qi, pj) with pi and pj on the boundary.

Crude Worst Case Time Analysis

• T0(n) = the running time of MinDisc(i) on n points
• T1(n) = nT1(n) + cn
• T2(n) = nT2(n) + cn
• By substitution
  T1(n) = O(n^2)
  T2(n) = O(n^2)

Better Analysis

• How often is MinDisc1 actually called in MinDisc0((p1, p2, ..., pn))?
• T0(n) ≤ T1(i1) + ... + T1(in) + cn
  Where MinDisc1 called on just these values.
• Let’s try to limit the number of calls to MinDisc1.

Randomized Incremental Algorithm

MinDisc0(p1, p2, ..., pn)
Randomly permute (p1, p2, ..., pn)
Let D0 be the smallest disc containing p1 and pn
For i = 3 to n do
  if pi in D0 then
    D0 := D0
  else
    D0 := MinDisc1((p1, p2, ..., pi-1, pi))
  Return D0
**Backwards Analysis**

- What is the probability that MinDisc1 is called.
- Fix $D_i$ to be the smallest disc containing $(p_1, p_2, ..., p_i)$
- Choose an element with equal probability to remove. What is the probability that smallest disc containing the remaining set is smaller? $\leq \frac{1}{3}$ because 3 boundary points determine $D_i$.
- The probability that MinDisc1($p_1, p_2, ..., p_{i-1}$, $p_i$) is called is $\frac{3}{i}$.

**Backward Analysis**

- By a similar argument, the probability that MinDisc2($p_1, p_2, ..., p_i$, $p$, $p$) is called is $\frac{2}{i}$.
- Expected time analysis of MinDisc1
  \[ E_i(n) \leq \frac{2}{2}T_2(2) + \cdots + \frac{2}{n}T_2(n) + cn \]
  \[ \leq \frac{2}{2}c2^1 + \cdots + \frac{2}{n}c2^1 + cn = 3cn \]
- Similar analysis of MinDisc0
  \[ E_i(n) \leq \frac{3}{3}E_2(3) + \cdots + \frac{3}{n}E_2(n) + cn \]
  \[ \leq \frac{3}{3}3c3^1 + \cdots + \frac{3}{n}3c3^1 + cn = 10cn \]

**Disc Uniqueness**

- All points are inside the circle centered at the midpoint between the centers of $D$ and $D'$.

  ![Disc Uniqueness Diagram](image)

**Disc Uniqueness**

- Circle of radius $d$ centered at midpoint contains all the points but is smaller.

  ![Disc Uniqueness Diagram](image)

**New Point Must Be on Boundary**

- If $p$ not in $D$ then $p$ on boundary of new smallest disc.

  ![New Point Must Be on Boundary Diagram](image)

**New Point Must be on Boundary**

- Move $D'$ toward center of $D$. The new disc is the same size and contains all the points. Contradicting uniqueness.

  ![New Point Must be on Boundary Diagram](image)
Notes on Randomized Incremental Algorithms

- Randomized Incremental Algorithms first used for computing the intersection of half-planes. (Seidel 1991)
- Application to smallest disc problem. (Welzl 1991)

k-d Tree

- Jon Bentley, 1975
- Tree used to store spatial data.
  - Nearest neighbor search.
  - Range queries.
  - Fast look-up
- k-d tree are guaranteed log₂ n depth where n is the number of points in the set.
  - Traditionally, k-d trees store points in d-dimensional space which are equivalent to vectors in d-dimensional space.

k-d Tree Construction

- If there is just one point, form a leaf with that point.
- Otherwise, divide the points in half by a line perpendicular to one of the axes.
- Recursively construct k-d trees for the two sets of points.
- Division strategies
  - divide points perpendicular to the axis with widest spread.
  - divide in a round-robin fashion.
k-d Tree Construction (10)

[Diagram showing k-d tree construction process]

k-d Tree Construction (11)

[Diagram showing k-d tree construction process]

k-d Tree Construction (12)

[Diagram showing k-d tree construction process]

k-d Tree Construction (13)

[Diagram showing k-d tree construction process]

k-d Tree Construction (14)

[Diagram showing k-d tree construction process]

k-d Tree Construction (15)

[Diagram showing k-d tree construction process]
**k-d Tree Construction (16)**

- First sort the points in each dimension.
  - $O(dn \log n)$ time and $dn$ storage.
  - These are stored in $A[1..d,1..n]$
- Finding the widest spread and equally divide into two subsets can be done in $O(dn)$ time.
- Constructing the k-d tree can be done in $O(dn \log n)$ and $dn$ storage.

**k-d Tree Construction (17)**

**k-d Tree Construction (18)**

**Node Structure for k-d Trees**

- A node has 5 fields
  - axis (splitting axis)
  - value (splitting value)
  - left (left subtree)
  - right (right subtree)
  - point (holds a point if left and right children are null)
k-d Tree Nearest Neighbor Search

\[
\text{NNS}(q, \text{point}, n, \text{node}, p, \text{ref point}, w, \text{ref distance}) \\
\text{if } n.\text{left} = n.\text{right} = \text{null} \text{ then (leaf case)} \\
\quad w^* := ||q - n.\text{point}||; \\
\quad \text{if } w < w^* \text{ then } w := w^*; p := n.\text{point}; \\
\text{else } \\
\quad \text{if } w = \text{infinity} \text{ then } \\
\quad \quad \text{if } q(\text{axis}) < n.\text{value} \text{ then } \\
\quad \quad \quad \text{NNS}(q, n.\text{left}, p, w); \\
\quad \quad \text{if } q(\text{axis}) + w > n.\text{value} \text{ then } \\
\quad \quad \quad \text{NNS}(q, n.\text{right}, p, w); \\
\quad \text{else (w is finite) } \\
\quad \quad \text{if } q(\text{axis}) - w < n.\text{value} \text{ then } \\
\quad \quad \quad \text{NNS}(q, n.\text{left}, p, w); \\
\quad \quad \text{if } q(\text{axis}) - w < n.\text{value} \text{ then } \\
\quad \quad \quad \text{NNS}(q, n.\text{right}, p, w); \\
\]

Initial call: \text{NNS}(q, \text{root}, p, \text{infinity})

**Explanation**

- **search left**
  - \(q(\text{axis}) - w < n.\text{value}\) means the circle overlap the left subtree.
- **search right**
  - \(q(\text{axis}) + w > n.\text{value}\) means the circle overlap the right subtree.

**Nearest Neighbor Search**

- Input \(q\)
- Find the leaf containing \(q\) and let \(w\) be the distance from the point in the leaf to \(q\).
- Search each subtree recursively that may have a point of distance \(< w\) to \(q\).
- Update \(w\) when a closer point is found.

**k-d Tree NNS (1)**

**k-d Tree NNS (2)**

**k-d Tree NNS (3)**
Notes on k-d Tree NNS

- K-d tree NNS been shown to run in $O(\log n)$ average time per search in a reasonable model. (Assume $d$ a constant)
  - Points come from the same distribution as the queries.
- Storage for the k-d tree is $O(n)$.
- Preprocessing time is $O(n \log n)$ assuming $d$ is a constant.

Computational Geometry Problems of Note

- Triangulation
- Binary Space Partition Trees
- Range queries
- Quad and Oct Trees
- Motion planning
- Paper Folding
- On-line Algorithms
- Kinetic Algorithms