Lecture 5: Divide & Conquer 1

2-D Maxima & Closest Pair

CS 341: Algorithms

Tuesday, Jan 22nd 2019
Outline For Today

1. 2-D Maxima
2. Closest Pair
Outline For Today

1. 2-D Maxima
2. Closest Pair
2-D Maxima

◆ Input: Set P of n 2-D points

◆ Output: All **maximal** points

◆ Dfn 1: Point p **dominates** point q iff
  - p.x > q.x AND p.y > q.y

◆ Dfn 2: Point p is **maximal** if no point dominates it
Example
Example

Q: Is $p_4$ maximal?
Q: Is $p_4$ maximal?

A: No. $p_2$, $p_3$, and $p_5$ dominate $p_4$. 
Example

Q: Is $p_2$ maximal?
Q: Is $p_2$ maximal?
A: No. $p_3$ dominates $p_4$. 
Example

Q: Is $p_5$ maximal?

A: Yes
Example

$p_3$, $p_5$, and $p_7$ are maximal

other points are not
Applications

◆ Databases: Skyline queries
  - Example “Skyline query”: minima
  - Find “minimal” hotels in a price & distance graph

◆ Economics: Finding “Pareto Optimal” points
procedure bruteForceMaxima(Set P of n points):
    M = {}
    for each p in P:
        for each q in P:
            check if p is dominated by q
        if p is not dominated:
            M.insert(p);
    return M

Runtime: O(n^2)
Can we Divide & Conquer?

- **Idea:** Let’s divide P vertically (on x-axis)
Can we Divide & Conquer?

◆ Idea: Let’s divide P vertically (on x-axis)

Q1: What can you say about a maximal point p in R?
A1: It is maximal in P as well.

Why?
No point in L can dominate p b/c: p.x ≥ x value of every point in L.
Q2: What can you say about a maximal point q in L?

A2: It’s maximal iff \( q.y \geq y \text{ value of every point in } R \).

In particular, let \( p^* \) be the point in R with max y.

Then q is maximal iff \( q.y \geq p^*.y \),
DC-Maxima

Procedure Algorithm(Set P of n points):
    Sort P by x values;  \(\text{O(nlog(n)) work}\)
    return DCMaxima(P)

Procedure DCMaxima (P sorted by x values):
    if (P.size == 1) return P;
    L = DCMaxima(P[1...n/2]);  \(\text{O(n) work}\)
    R = DCMaxima(P[n/2+1...n]);  \(\text{O(n) work}\)
    let p* be max y valued point in R;
    let M = R;
    for each q in L:
        if (q.y \geq p*.y)
            M.insert(q);
    return M;

Total: O(n) work outside recursive calls
Runtime Analysis

Recursive part: $T(n) = 2T(n/2) + O(n)$

By Master Thm: $O(n\log(n))$

Total Work:

1. Initial Sorting: $O(n\log(n))$
2. Recursive part: $O(n\log(n))$

Total: $O(n\log(n))$
Exercise

After initial sorting by x-axis, design an $O(n)$ time algorithm (not DC) that gives all of the maximal points.

Note: Total time still $O(n\log(n))$ b/c of sorting. But post-sorting work is linear instead of $O(n\log(n))$ of DCMaxima.

Fact: $\Omega(n\log(n))$ lower bound for comparison based algs.
Outline For Today

1. 2-D Maxima

2. Closest Pair
Closest Pair Problem

- Input: Set P of n 2-D points
- Output: pair p and q s.t. dist(p, q) minimum over all pairs
- Break ties arbitrarily
- dist(p, q): Euclidean distance

\[ \text{dist}(p, q) = \sqrt{(p.x - q.x)^2 + (p.y - q.y)^2} \]
Example

\[ p_5 = (-5, 3) \]

\[ p_4 = (0, 1) \]

\[ p_3 = (6, 5) \]

\[ p_2 = (1, 2) \]

\[ p_1 = (4, -3) \]
Example

Closest pair: \((p_2, p_4)\); 
\[\text{dist}(p_2, p_4) = \sqrt{1^2 + 1^2} = \sqrt{2}\]
Applications

- Very fundamental computational problem
  - Databases
  - Machine Learning
  - Image Processing
  - Computational Geometry
1-D Version

- \((x_1, x_2, \ldots, x_n) = (2.2, 5.8, 1.1, -3.0, 1.2, \ldots)\)
- Just sort and scan:
  - compare each point with the next point in the sorted order
  - b/c closest pair has to be a consecutive pair
- Sort: \(O(n\log(n))\) time
- Scan: \(O(n)\) time
- Total: \(O(n\log(n))\) time

**Question:** Can we do \((n\log(n))\) in 2-D?
Alg 1: Brute Force

procedure bruteForceCP(Set P of n points):
    minPair = {}
    minDist = +∞
    for each p in P:
        for each q in P:
            if (dist(p, q) < minDist)
                minPair.insert(p, q);
                minDist = dist(p, q)
    return minPair

Runtime: O(n²)
Can we Divide & Conquer?

◆ Same idea as maxima: Divide P on x-axis

Claim that doesn’t require a proof: closest pair \((p, q)\):

1. \((p, q)\) both in \(L\);
2. \((p, q)\) both in \(R\); or
3. \(p\) is in \(L\) and \(q\) is in \(R\)
procedure Algorithm(P of n points):
    sort P by x values
    DC-CP(P)

procedure DC-CP(P sorted by x values):
    if (P.size ≤ 3) compare all & return closest;
    pair_L = DC-CP(P[1,...,n/2])
    pair_R = DC-CP(P[n/2+1,...,n])
    pair_S = findMinSpanningPair(P)
    return min(pair_L, pair_R, pair_S)

Q: How can we find the spanning pair quickly?
Observation 1

Let $\delta = \min (\text{dist}(\text{pair}_L), \text{dist}(\text{pair}_R))$

Then pair $s$ (if closest globally) lies in the above $2\delta$-wide green strip

Q: Why?
Q: Can $p_5$ be part of a globally closest pair? 
A: No. Any $p$ in $R$ is already has dist > $\delta$ to $p_5$. 
Observation 2

- Say, $p_7$ (the lowest y valued point in strip) is in pair $s$

- Then the other point can only lie in this $\delta \times \delta$ square.

$Q$: Why?

*Has to be on the opposite side & can’t be $> \delta$ higher than $p_7$ on y axis.*
Core Idea For Finding Spanning Pair

1. Start from lowest $y$ valued point in the strip
2. Search the $\delta \times \delta$ square points on the opposite side
3. Repeat 1 & 2 this for the next lowest $y$-valued point
4. So on and so forth...
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point in the strip
2. Search the $\delta \times \delta$ square points on the opposite side
3. Repeat 1 & 2 for the next lowest y-valued point
4. So on and so forth...
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point in the strip
2. Search the \( \delta \times \delta \) square points on the opposite side
3. Repeat 1 & 2 this for the next lowest y-valued point
4. So on and so forth...
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point in the strip
2. Search the $\delta \times \delta$ square points on the opposite side
3. Repeat 1 & 2 this for the next lowest y-valued point
4. So on and so forth...
Core Idea For Finding Spanning Pair

1. Start from lowest y valued point in the strip
2. Search the $\delta \times \delta$ square points on the opposite side
3. Repeat 1 & 2 this for the next lowest y-valued point
4. So on and so forth...
A More Practical Idea

- Don’t differentiate between same and opposite side
- Just search the $2\delta \times \delta$ above rectangle each time
Don’t differentiate between same and opposite side
Just search the $2\delta \times \delta$ above rectangle each time
Don’t differentiate between same and opposite side
Just search the $2\delta \times \delta$ above rectangle each time
A More Practical Idea

- Don’t differentiate between same and opposite side
- Just search the $2\delta \times \delta$ above rectangle each time
procedure Algorithm(P of n points):
    sort P by x values
    DC-CP1(P)

procedure DC-CP1(P sorted by x values):
    if (P.size ≤ 3) compare all & return closest;
    pair_L = DC-CP1(P[1,...,n/2])
    pair_R = DC-CP1(P[n/2+1,...,n])
    δ = min(dist(pair_L, pair_R))
    pair_S = findMinSpanningPair(δ, P)
    return min(pair_L, pair_R, pair_S)
procedure findMinSpanningPair (δ, P):
S = select each p in P s.t |p_{n/2}.x-p.x| \leq δ \rightarrow O(n)
sort(S by increasing y values) \rightarrow O(n\log(n))
minDist = +\infty
minPair = null;
for i = 1 to S.length: \rightarrow O(n)
    j = i+1 (compare S[i] to only the points above S[i])
    while (|S[j].y - S[i].y| \leq \delta):
        if (dist(S[i], S[j]) < minDist):
            minPair = (S[i], S[j]);
            minDist = dist(S[i], S[j])
    j++;
return minPair

Q: How many times does the while loop execute? 
Claim: O(1) times
For a point $p$, how many times does while loop execute?

**Obs:** as many times as there are points in the $2\delta \times \delta$ rectangle.

**Q:** How many points can be in a $2\delta \times \delta$ rectangle?

**A:** As many as in the left $\delta \times \delta$ square + right $\delta \times \delta$ square.
Recall: Each point in the square is at least at distance $\delta$.

Q1: How many can fit the lower triangle?

A: 3

no other point can be inside the triangle except the other two corners
# Points in a $\delta \times \delta$ Square

Recall: Each point in the square is at least at distance $\delta$.

Q1: How many can fit the lower triangle?
   A: 3

Q2: How many can fit the square?
   A: 4
For a point $p$, how many times does while loop execute?

**Obs:** as many times as there are points in the $2\delta \times \delta$ rectangle.

# points in the $2\delta \times \delta$ rectangle $\leq 4 + 4 = 8$
procedure findMinSpanningPair (δ, P):
S = select each p in P s.t |P[n/2].x-p.x| ≤ δ 
sort(S by increasing y values) \(\Rightarrow O(n\log(n))\)
minDist = +∞ 
minPair = null;
for i = 1 to S.length: \(\Rightarrow O(n)\)
    j = i+1
    while (|S[j].y - S[i].y| ≤ δ):
        if (dist(S[i], S[j]) < minDist) {
            minPair = (S[i], S[j])
        }
        j++;
return minPair

Total: \(O(n\log(n))\)
DC-CP 1: Runtime Analysis (1)

**procedure** DC-CP1(P sorted by x values):

if (P.size \( \leq 3 \)) compare all & return closest;

\( \text{pair}_L = \text{DC-CP1}(P[1, \ldots, n/2]) \)

\( \text{pair}_R = \text{DC-CP1}(P[n/2+1, \ldots, n]) \)

\( \delta = \min(\text{dist}(\text{pair}_L, \text{pair}_R)) \)

\( \text{pair}_S = \text{findMinSpanningPair}(\delta, P) \)

**return** \( \min(\text{pair}_L, \text{pair}_R, \text{pair}_S) \)

**Recursive part**: Outside Recursive Calls: \( n \log(n) \) work.

\[ T(n) = 2T(n/2) + n \log(n) \]

**Exercise**: Show by induction or recursion tree that total work of recursive part is \( O(n \log^2(n)) \).

**Total Alg Work**: \( O(n \log(n)) + O(n \log^2(n)) = O(n \log^2(n)) \).

**Can improve to** \( O(n \log(n)) \) **by pre-sorting** \( P \) **also by** \( y \).
Shamos’ DC Algorithm (1975) (1)

```plaintext
procedure Algorithm(P of n points):
P_x = sort P by x values in increasing order
P_y = sort P by y values in increasing order
DC-Shamos(P_x, P_y)

procedure DC-Shamos(P_x, P_y):
  if (P_x.size ≤ 3) ...;
P_yL = select from P_y points with x ≤ P_x[n/2].x
P_yR = select from P_y points with x > P_x[n/2].x
pair_L = DC-Shamos(P_x[1,...,n/2], P_yL)
pair_R = DC-Shamos(P_x[n/2+1,...,n] P_yR)
δ = min(dist(pair_L, pair_R))
pair_s = findMinSpanningPairShamos(δ, P_x, P_y)
return min(pair_L, pair_R, pair_s)
```

Sorted by y already!
procedure findMinSpanningPairShamos(\(\delta, P_x, P_y\)):
S = select each \(p\) in \(P_y\) s.t \(|P_x[n/2].x-p.x| \leq \delta\)
minDist = +\(\infty\)
minPair = null;
for \(i = 1\) to \(S\.length\): \(\mathcal{O}(n)\)
    \(j = i+1\)
    while \(|S[j].y - S[i].y| \leq \delta\):
        if (dist(S[i], S[j]) < minDist):
            minPair = (S[i], S[j])
            \(j++\);
return minPair

\(\text{Total: } \mathcal{O}(n)\)
Runtime Analysis of Shamos’ Algorithm

Key Idea of Shamos: Avoid sorting by y values in each recursive call by pre-sorting P by y values.

Recursive part: Outside Recursive Calls: $O(n)$ work.

\[
T(n) = 2T(n/2) + O(n)
\]

By Master Thm, total: $O(n \log(n))$

Total Work for Shamos

$O(n \log(n)) + O(n \log(n)) = O(n \log(n))$. 