CS 341: ALGORITHMS

Lecture 5: finishing D&C, greedy algorithms I

Readings: see website

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca
THE CLOSEST PAIR PROBLEM

classroom

overworked student

When someone near you

COUGHS
THE CLOSEST PAIR PROBLEM

◆ Input: Set $P$ of $n$ 2D points

◆ Output: pair $p$ and $q$ s.t. $\text{dist}(p, q)$ minimum over all pairs

◆ Break ties arbitrarily

◆ $\text{dist}(p, q) = \sqrt{(p.x - q.x)^2 + (p.y - q.y)^2}$
Can we Divide & Conquer?

◆ Like non-dominated points: sort by x-axis & divide in half

\[ p_2 \circ \quad p_4 \quad p_7 \circ \]
\[ p_1 \circ \quad p_5 \quad p_6 \circ \]
\[ p_3 \quad p_8 \circ \]
\[ \text{L} \quad \text{R} \]

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

1. \((p, q)\) both in L or
2. \((p, q)\) both in R or
3. One of \((p, q)\) in L and one of \((p, q)\) in R

We call this a spanning pair
How to efficiently compute the minimum spanning pair?
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 $p$ be part of a globally closest spanning pair?  
A: No. Everything in $R$ has $\text{dist} > \delta$ to $p$. And we already have a solution with $\text{dist} = \delta$. 
Observation 2

◆ Say, \( p \) (the lowest y valued point in strip) is in pair\(_s\).

\[
\delta \quad \delta
\]

\(\text{Has to be on the opposite side & can’t be } > \delta \text{ higher than } p \text{ on y axis.}\)

\(\text{Q: Why?}\)

◆ Then the other point can only lie in this \(\delta \times \delta\) square.
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 for the next lowest y-valued point
4. So on and so forth...

---

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$\delta$</th>
</tr>
</thead>
</table>

---

$\leftarrow L \rightarrow R \rightarrow$
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 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...

Switching sides might complicate code...
Turns out it’s not needed to get good time complexity.
A More Practical Idea

◆ 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
A More Practical Idea

◆ 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
ClosestPair(P[1..n])
sort(P) by x values
Recurse(P)

Recurse(P[1..n]) // precondition: P sorted by x
  // base case
  if n < 4 then compare all pairs and return closest

  // divide & conquer
  pairL = Recurse(P[1..(n/2)])
  pairR = Recurse(P[(n/2)+1..n])

  // combine
  δ = min(dist(pairL), dist(pairR))
  pairS = findMinSpanningPair(P, δ)
  return minDistPair(pairL, pairR, pairS)
Claim: inner loop performs $O(1)$ iterations!
For a particular $i$, how many $j$ iterations occur?

Obs: as many 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.
POINTS IN A $\delta \times \delta$ SQUARE

- Recall $\delta$ is the smallest distance between any pair of points that are both in $L$ or both in $R$.
- Note this square is entirely in $L$ or entirely in $R$.

So, $\delta$ is the smallest distance between any pair of points in this square!

A point in the middle would rule out any other points.

So, most efficient packing of points puts one in each corner (4 total).
For a particular $i$, how many $j$ iterations occur?

**Obs:** as many 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.

Can only contain eight points! (technically six)
\textbf{Time complexity (unit cost)}

```python
findMinSpanningPair(\delta, P[1..n])  // P sorted by x
S = \{ p in P : abs(P[n/2].x - p.x) \leq \delta \}
sort(S) by increasing y values
if |S| < 2 return (-\infty, -\infty), (\infty, \infty)
minPair = (S[1], S[2])  // arbitrary pair to start
for i = 1..len(S)
    for j = (i+1)..len(S)
        if S[j].y - S[i].y > \delta then break
        minPair = minDistPair(minPair, (S[1], S[j]))
return minPair
```

- \textit{j}-loop performs at most \textbf{eight} iterations
- Each does \(\Theta(1)\) work, so entire \textbf{j}-loop does \(\Theta(1)\) work!
- So entire \textbf{i}-loop does \(\Theta(n)\) work
- So, findMinSpanningPair does \(\Theta(n \log n)\) work
Time complexity (unit cost)

- \( T'(n) \): ClosestPair\((P[1..n])\)
- \( T(n) \): Recurse\((P[1..n])\)
- \( T'(n) = \Theta(n \log n) + T(n) \)
- \( T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n \log n) \)
- Lec2 notes using recursion trees showed
  - \( T(n) \in \Theta(n \log^2 n) \)
  - \( T'(n) \in \Theta(n \log n) + \Theta(n \log^2 n) \)
  - So \( T'(n) \in \Theta(n \log^2 n) \)
IMPROVING THIS RESULT FURTHER
IMPROVING THE PREVIOUS ALGORITHM

- Sorting by $y$-values causes findMinSpanningPair to take $O(n \log n)$ time instead of $O(n)$ time.
- This happens in each recursive call, and dominates the running time.
- Avoid sorting $P$ over and over by creating another copy of $P$ that is pre-sorted by $y$-values.
- Assume for simplicity that $x$ coordinates are unique.
Shamos algorithm (1975)

This selection step preserves the y-sort order

Observe $PxL$ and $PyL$ contain the same points
(specifically the points with $x \leq x_{mid}$)

Moreover $PxL$ is sorted by $x$ while $PyL$ is sorted by $y$

And similarly for $PxR$, $PyR$...

No need to sort in Recurse!

```
ShamosClosestPair(P[1..n])
 Px = sort(P) by increasing x values
 Py = sort(P) by increasing y values
 Recurse(Px, Py)

Recurse(Px[1..n], Py[1..n])
 // base case
 if n < 4 then return BruteForce(Px)

 // divide & conquer
 xmid = Px[n/2].x
 PxL = Px[1..(n/2)] // x <= xmid
 PxR = Px[(n/2+1)..n] // x > xmid
 PyL = select p from Py where p.x <= xmid
 PyR = select p from Py where p.x > xmid
 pairL = Recurse(PxL, PyL)
 pairR = Recurse(PxR, PyR)

 // combine
 δ = min(dist(pairL), dist(pairR))
 pairS = findMinSpanningPair(δ, Py, xmid)
 return minDistPair(pairL, pairR, pairS)
```
findMinSpanningPair(\(\delta\), Py[1..n], xmid) \ // Py sorted by y
S = \{ p in Py : abs(xmid - p.x) <= \(\delta\) \}
if |S| < 2 return (-\(\infty\), -\(\infty\)), (\(\infty\), \(\infty\))
minPair = (S[1], S[2]) \ // arbitrary pair to start
for i = 1..len(S)
  for j = (i+1)..len(S)
    if S[j].y - S[i].y > \(\delta\) then break
    minPair = minDistPair(minPair, (S[i], S[j]))
return minPair

\(\Theta(n)\) and preserves the y-sort order

\(\Theta(n)\)

Total \(\Theta(n)\) for this function
ShamosClosestPair(P[1..n])
   Px = sort(P) by increasing x values
   Py = sort(P) by increasing y values
   Recurse(Px, Py)

Recurse(Px[1..n], Py[1..n])
   // base case
   if n < 4 then return BruteForce(Px)

   // divide & conquer
   xmid = Px[n/2].x
   PxL = Px[1..(n/2)] // x <= xmid
   PxR = Px[(n/2+1)..n] // x > xmid
   PyL = select p from Py where p.x <= xmid
   PyR = select p from Py where p.x > xmid
   pairL = Recurse(PxL, PyL)
   pairR = Recurse(PxR, PyR)

   // combine
   δ = min(dist(pairL), dist(pairR))
   pairS = findMinSpanningPair(δ, Py, xmid)
   return minDistPair(pairL, pairR, pairS)

Time complexity

\[ T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \]

So runtime for Shamos' algorithm is in \( \Theta(n \log n) \)
GREEDY ALGORITHMS
Optimization Problems

**Problem**: Given a problem instance, find a feasible solution that maximizes (or minimizes) a certain objective function.

**Problem Instance**: Input for the specified problem.

**Problem Constraints**: Requirements that must be satisfied by any feasible solution.

**Feasible Solution**: For any problem instance $I$, $\text{feasible}(I)$ is the set of all outputs (i.e., solutions) for the instance $I$ that satisfy the given constraints.

**Objective Function**: A function $f : \text{feasible}(I) \rightarrow \mathbb{R}^+ \cup \{0\}$. We often think of $f$ as being a profit or a cost function.

**Optimal Solution**: A feasible solution $X \in \text{feasible}(I)$ such that the profit $f(X)$ is maximized (or the cost $f(X)$ is minimized).
SOLVING OPTIMIZATION PROBLEMS

- Lots of techniques
- We will study **greedy** approaches first
- Later, dynamic programming
  - Sort of like divide and conquer but can **sometimes** be much more efficient than D&C
- Greedy algorithms are usually
  - Very fast, but hard to prove optimality for
  - Structured as follows…
The Greedy Method

partial solutions

Given a problem instance $I$, it should be possible to write a feasible solution $X$ as a tuple $[x_1, x_2, \ldots, x_n]$ for some integer $n$, where $x_i \in \mathcal{X}$ for all $i$. A tuple $[x_1, \ldots, x_i]$ where $i < n$ is a partial solution if no constraints are violated. Note: it may be the case that a partial solution cannot be extended to a feasible solution.

choice set

For a partial solution $X = [x_1, \ldots, x_i]$ where $i < n$, we define the choice set

$$\text{choice}(X) = \{y \in \mathcal{X} : [x_1, \ldots, x_i, y] \text{ is a partial solution}\}.$$
The Greedy Method (cont.)

local evaluation criterion

For any $y \in X$, $g(y)$ is a **local evaluation criterion** that measures the cost or profit of including $y$ in a (partial) solution.

extension

Given a partial solution $X = [x_1, \ldots, x_i]$ where $i < n$, choose $y \in \text{choice}(X)$ so that $g(y)$ is as small (or large) as possible. Update $X$ to be the $(i + 1)$-tuple $[x_1, \ldots, x_i, y]$.

greedy algorithm

Starting with the “empty” partial solution, repeatedly extend it until a feasible solution $X$ is constructed. This feasible solution may or may not be optimal.

Local evaluation means we **cannot** consider future choices when deciding whether to include $y$ in our solution.

We **irrevocably** decide to include $y$ (or not). We do not reconsider.

We choose the next element to include **greedily** by taking the $y$ that gives the **maximum local improvement**.

This may or may not be a good idea...
CORE CHARACTERISTICS
OF GREEDY ALGORITHMS

Greedy algorithms do no **looking ahead** and no **backtracking**.

Greedy algorithms can usually be implemented efficiently. Often they consist of a **preprocessing step** based on the function $g$, followed by a **single pass** through the data.

In a greedy algorithm, only **one feasible solution** is constructed.

The execution of a greedy algorithm is based on **local criteria** (i.e., the values of the function $g$).

**Correctness:** For certain greedy algorithms, it is possible to prove that they always yield optimal solutions. However, these proofs can be tricky and complicated!
PROBLEM: INTERVAL SELECTION
PROBLEM: INTERVAL SELECTION

- **Input:** a set $A = \{A_1, ..., An\}$ of time intervals
  - Each interval $A_i$ has a start time $s_i$ and a finish time $f_i$

- **Feasible solution:** a subset $X$ of $A$ containing **pairwise disjoint** intervals

- **Output:** a feasible solution of **maximum size**
  - i.e., one that maximizes $|X|$}

Where $s_i$ and $f_i$ are positive integers

Chosen

Rejected

Bad solution. Not optimal!
POSSIBLE GREEDY STRATEGIES

1. Sort the intervals in increasing order of starting times. At any stage, choose the earliest starting interval that is disjoint from all previously chosen intervals

- **Partial solutions**
  - $X = [x_1, x_2, ..., x_i]$ where each $x_i$ is an interval for the output

- **Choices**
  - $X = A$ (i.e., all intervals)
  - Choice$(X) = \{ y \in X : [x_1, ..., x_i, y] \text{ respects all constraints} \}$
    - i.e., where $y \notin X$ and $\forall x \in X \text{ disjoint}(y, x)$

- **Local evaluation function**
  - $g(y) = s_j$ where $y = A[j]$
  - (i.e., $g(y)$ = start time of interval $y$)
POSSIBLE GREEDY STRATEGIES
FOR INTERVAL SELECTION

1. Sort the intervals in increasing order of *starting times*. At any stage, choose the *earliest starting* interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $s_i$).

2. Sort the intervals in increasing order of *duration*. At any stage, choose the interval of *minimum duration* that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i - s_i$).

3. Sort the intervals in increasing order of *finishing times*. At any stage, choose the *earliest finishing* interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i$).

Does one of these strategies yield a *correct* greedy algorithm?
STRATEGY 1: PROVING INCORRECTNESS

- Idea: find **one input** for which the algorithm gives a **non-optimal solution** or an **infeasible solution**

Sort the intervals in increasing order of **starting times**. At any stage, choose the **earliest starting** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $s_i$).

**Strategy 1**

Consider input: $[0, 10), [1, 3), [5, 7)$.
HOW ABOUT STRATEGY 2?

Strategy 2: Sort the intervals in increasing order of duration. At any stage, choose the interval of minimum duration that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i - s_i$).

Consider input: $[0, 5), [6, 10), [4, 7)$.

![Diagram showing intervals and x-axis]

We will show that Strategy 3 (sort in increasing order of finishing times) always yields the optimal solution.
GreedyIntervalSelection(A[1..n])
  sort(A) by increasing finish times
  X = [A[1]]
  prev = 1       // index of last selected interval
  for i = 2..n
      if A[i].s >= A[prev].f then
          X.append(A[i])
          prev = i
  return X

Where is our local evaluation function $g$ in this code?
GreedyIntervalSelection(A[1..n])
   sort(A) by increasing finish times
   X = [A[1]]
   prev = 1    // index of last selected interval
   for i = 2..n
      if A[i].s >= A[prev].f then
         X.append(A[i])
         prev = i
   return X

How to prove this is correct?
(I.e., how can we show the returned solution is both feasible and optimal?)

Feasibility? Easy!
We always choose an interval that starts after all other chosen intervals end

Optimality? Harder…
BRO I DON'T WANT PROOF

I WANT EVIDENCE
GREEDY CORRECTNESS PROOFS

- Want to prove: greedy solution $X$ is correct (feasible & optimal)
- Usually show **feasibility directly** and **optimality by contradiction**:
  - Suppose solution $O$ is better than $X$
  - Show this necessarily leads to a contradiction
- Two broad strategies for deriving this contradiction:
  1. **Greedy stays ahead**: show every choice in $X$ is “at least as good” as the corresponding choice in $O$
  2. **Exchange**: show $O$ can be improved by replacing some choice in $O$ with a choice in $X$

Let’s demonstrate approach #1 (next time)