CS 341: ALGORITHMS
Lecture 5: finishing D&C, greedy algorithms
Readings: see website
Trevor Brown
https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca

THE CLOSEST PAIR PROBLEM

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

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

Observation 1

◆ Let δ = min (dist(pair_L), dist(pair_R))
◆ Then pair_s (if closest globally) lies in the above 2δ-wide green strip
Q: Why?
Example for Observation 1

Q: Can p be part of a globally closest spanning pair s?
A: No. Everything in R has dist > δ to p.
And we already have a solution with dist = δ.

Observation 2

◆ Say, p (the lowest y valued point in strip) is in pair s
◆ Then the other point can only lie in this δ×δ square.

Q: Why?
Has to be on the opposite side & can’t be > δ higher than p on y axis.

Core Idea For Finding Spanning Pair

1. Start from lowest y valued point in the strip
2. Search the δ×δ 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
Claim: inner loop performs $O(1)$ iterations!

**Points in a δ × δ Square**

- Recall δ 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, δ 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).

If $|S| < 2$ return $-\infty$, $-\infty$, $(\infty, \infty)$

---

**For a particular $i$, how many $j$ iterations occur?**

**Notes:**

- Points in $S$ \(\Theta(n)\)
- Time complexity? \(\Theta(n \log n)\)
- \(\Theta(1)\)?
- \(\Theta(1)\)?
- \(\Theta(1)\)?

**Time complexity (unit cost)**

- $\delta$ loop performs at most eight iterations
- Each does $\Theta(1)$ work, so entire $\delta$ loop does $\Theta(1)$ work!
- So entire $\delta$ loop does $\Theta(n)$ work
- So, `findMinSpanningPair` does $\Theta(n \log n)$ work
IMPROVING THIS RESULT FURTHER

IMPROVING THE PREVIOUS ALGORITHM

- Sorting by $y$-values causes $\text{findMinSpanningPair}$ 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 $P_L$ and $P_R$ contain the same points (specifically the points with $x \leq x_{\text{mid}}$).
- Moreover $P_L$ is sorted by $x$ while $P_R$ is sorted by $y$.
- No need to sort in $\text{Recurs}()$.

Time complexity

- $T(n) = 2T(\frac{n}{2}) + O(n)$
- Merge sort recurrence, $\Omega(n \log n)$
- So runtime for Shamos' algorithm is $\Theta(n \log n)$.
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 $x_i \in X$ 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
  
  $$\text{choice}(X) = \{ y \in X : [x_1, \ldots, x_i, y] \text{ is a partial solution} \}.$$

**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

- **Input:** a set $A = (A_1, \ldots, A_n)$ 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|

POSSIBLE GREEDY STRATEGIES

- **Partial solutions** $X = [x_1, x_2, \ldots, x_i]$ where each $x_i$ is an interval for the output
- **Choices** $X = A$ (i.e., all intervals)
  - $X = Y \in X : [x_1, \ldots, x_i, y]$ respects all constraints
    - i.e., where $y \notin X$ and $y \notin X$ (deletion)
- **Local evaluation function** $g(y) = s_j$ where $y = A[j]$
  - (i.e., $g(y)$ = start time of interval $y$)

STRATEGY 1: PROVING INCORRECTNESS

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

HOW ABOUT STRATEGY 2?

We will show that Strategy 3 (sort in increasing order of finishing times) always yields the optimal solution.

STRATEGY 3

- **Local evaluation function** $g$ in this code?
**STRATEGY 3**

**Time complexity:** Sort + one pass \( \in \Theta(n \log n) \)

---

**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...**

---

**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

---

**Lemma 4.2 (Greedy stays ahead)**

\( f_i < f_j \) for \( m = 1, 2, \ldots \)

**Proof**

- **Initial case:** \( m = 1 \). We have \( f_1 \leq f_2 \) since the greedy algorithm begins by choosing \( s_1 \). \( A_1 \) has the earliest finishing time.
- **Induction assumption:** \( f_{i-1} \leq f_{j-1} \). Consider \( A_i \) and \( A_j \). We have \( s_{i-1} \geq f_{i-1} \geq f_{j-1} \) (by I.H.).
- \( f_{i-1} \geq f_{j-1} \) (by I.H.).
- \( A_i \) has the earliest finishing time of any interval that starts after \( f_{i-1} \). Therefore \( f_i \leq f_{i-1} \).

---

**Correctness Proof (cont.)**

**Recall**

Greedy solution is \( X = (A_1, \ldots, A_k) \).

Optimal solution is \( O = (A_{j_1}, \ldots, A_{j_l}) \).

We give an induction proof.

Let \( X \) be the greedy solution,

\[ X = (A_{i_1}, \ldots, A_{i_k}) \]

where \( j_1 < \cdots < j_l \).

Let \( O \) be any optimal solution,

\[ O = (A_{j_1}, \ldots, A_{j_l}) \]

where \( j_1 < \cdots < j_l \).

We are merely imagining reordering the intervals chosen by the optimal algorithm so we can easily compare their finish times to intervals in \( X \).

**We are NOT assuming the optimal algorithm uses the same sort order!**

---

**CRUCIAL**

- \( i, j \) are subsequences of the sorted intervals
- To obtain a contradiction
  1. \( A_{i_k} \) starts before \( A_{j_1} \) by induction.
  2. \( A_{j_1} \) finishes before \( A_{i_k} \) by lemma.
  3. \( A_{i_k} \) starts after \( A_{j_1} \) finishes.

This completes the proof!
A DIFFERENT PROOF

"Slick" ad-hoc approaches are sometimes possible...

Let \( F = \{ f_1, \ldots, f_n \} \) be the finishing times of the intervals in \( X \).

No interval finishes strictly to the left.

No interval starts strictly to the right.

No interval in \( X \) is strictly between these points!

So, in addition to the intervals in \( X \), only the following types of intervals are possible:

- Contains \( f_i \)
- Contains \( f_j \)
- Contains \( f_i \) and \( f_j \)

Thus, every interval contains some finishing time in \( F \).

And, two intervals in \( F \) cannot contain the same element of \( F \).

So, there must be as many finishing times in \( F \) as there are intervals in \( X \). QED