CSE 241 Class 2

Jeremy Buhler

August 26, 2015

The following three sections introduce the divide-and-conquer algorithm for closest pair. Why? I want to show you an interesting, nontrivial algorithm and how we analyze it.

We *compute distance only*. Of course, you can save points whenever min distance is updated, just as for naive algorithm.

1 Algorithm Part 1: Preprocessing

- Recall that P is input array of n points. [**Sketch five points in space as shown at right**]
- Create **two sorted arrays of references to points in** P. Note that arrays refer to *same points*, just in different orders.
  - **ptsByX** enumerates points of P in increasing order by x.
  - **ptsByY** enumerates points of P in increasing order by y.

  [**Embellish the five points with ptsByX, ptsByY illustrations:**]

- Algorithm ClosestPair takes sorted arrays **ptsByX**, **ptsByY**, and input size n.

2 Algorithm Part 2: Divide-and-Conquer Skeleton

The **divide-and-conquer strategy**:

- split large problem into smaller parts (**divide**)
- solve the smaller parts recursively (**conquer**) (maintain *invariant*: parts must be sorted like original problem!)

- combine smaller solutions (**combine**)
- Can be much faster than solving entire problem at once

```
ClosestPair(ptsByX, ptsByY, n)
  if n = 1
     return ∞
  if n = 2
     return distance(ptsByX[0], ptsByX[1])

  mid ← ⌈n/2⌉ − 1                                   ▷ divide into two subproblems
  copy ptsByX[0 ... mid] into new array XL in x order.
  copy ptsByX[mid+1 ... n − 1] into new array XR in x order.

  copy ptsByY into arrays YL and YR in y order, s.t.
      XL and YL refer to same points, as do XR and YR

  distL ← ClosestPair(XL, YL, ⌈n/2⌉)                  ▷ conquer
  distR ← ClosestPair(XR, YR, ⌊n/2⌋)

  return Combine(ptsByY, ptsByX[mid], n, min(distL, distR))
```

[At points where we introduce XL, XR and YL, YR, add them to the diagram like this:]

3 Algorithm Part 3: Combine Step

To combine, must consider pairs of points that cross the dividing line in x.

```
Combine(ptsByY, midPoint, n, lrDist)
   construct array yStrip, in increasing y order, of all points p in
        ptsByY s.t. |p.x − midPoint.x| < lrDist
   minDist ← lrDist
   for j in 0 ... yStrip.length − 2 do
       k ← j + 1
       while k ≤ yStrip.length − 1 and yStrip[k].y − yStrip[j].y < lrDist do
           d ← distance(yStrip[j], yStrip[k])
           minDist ← min(minDist, d)
           k++
   return minDist
```

[Illustrate difficult first step on diagram, including the center STRIP]

4 Correctness

In divide-and-conquer algorithms, correctness proofs are generally by induction on input size n.

- **Base case**: trivially correct for $n = 1, 2$
- **Inductive case**: Assume distL and distR are min. pairwise dists between points on either side of partition (size $< n$).
- If both points in closest pair are on same side, no pairs checked by Combine can change **minDist**, and we are done.

- If points (p, q) in closest pair are on opposite sides ...
  - p and q at distance less than **lrDist**
  - $p.x$ and $q.x$ must be within **lrDist** of each other; hence, each is within **lrDist** of partition line in x
  - Moreover, $p.y$ and $q.y$ must be with **lrDist** of each other, so are found by **while** loop.
- Hence, closest pair is always found for size n. QED

5 Cost, Part 1

What is worst-case running time of ClosestPair on inputs of size n? Let's try *statement counting* (without being too careful about constants):

[Do following counts and defns on overhead of algorithm.]

- call $T(n)$ the running time of the algorithm on input of size n
- base case takes constant time c_0
- creating XL, YL, XR, YR takes time $c_1 n$ (DO NOT SORT!)
- creating array **yStrip** takes time $c_2 n$ (DO NOT SORT!)
- *what about recursive calls?* Let's write costs implicitly: $T(\lceil n/2 \rceil)$ and $T(\lfloor n/2 \rfloor)$
- *what about* COMBINE*?* Outer loop statements are $c_3 n + c_4$.
- *Inner **while** loop?* Naively, seems it could run **yStrip.length** $-j-1$ times????

Now for the cool part!
(back to board)

6 Cost, Part 2

Claim: inner loop of COMBINE never runs more than seven times.

- Consider loop execution for any point yStrip[j].
- Each loop iteration handles a distinct point yStrip[k] inside a box of size 2 lrDist wide by lrDist high
- Any two points on same side of partition are at least lrDist apart!

Lemma: (geometry) you can't fit five points in a $\delta \times \delta$ box *and* have every pair be at distance at least δ.

[Draw box diagram on board:]

- "Obvious" that two points in a $\delta/2$ by $\delta/2$ box are always at distance less than δ.
- Divide box into four quarters, and throw five points into box. By *pigeonhole principle*, some quarter contains two points.
- Hence, not all pairs in box at distance $\geq \delta$. QED

finish the proof

- Left and right halves of big box are **lrDist** by **lrDist**
- Hence, each half contains at most four points (else some pair on same side would be closer than **lrDist**).
- Conclude that box contains only eight points, *including* yStrip[j]. QED

7 Cost, Part 3

OK, we've filled in missing inner loop time. Conclude that...

$$T(n) = \begin{cases} c_0 & \text{if } n \leq 2 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn + d & \text{if } n > 2 \end{cases}$$

- Assume n is power of two for simplicity. Also, increase c until linear term dominates constant term. This can only increase our time estimate, so no harm done.

$$T(n) \leq \begin{cases} c_0 & \text{if } n \leq 2 \\ 2T(n/2) + c'n & \text{if } n > 2 \end{cases}$$

- This is a **recurrence** for time $T(n)$: **a definition of $T(n)$ in terms of $T(n')$, for** $n' < n$.
- How do we solve this recurrence to find $T(n)$ in terms of n? Detailed discussion later, but here's a good graphical method: the **recursion tree**.

1. How many levels in tree? Each time, we divide n by 2, so to reach 2 (the base case), we need $\log_2 n$ levels.

2. On kth level (root has $k = 0$), input size is $n/2^k$, so we do $c'n/2^k$ work per node (besides recurring). But there are 2^k nodes on level k, so . . .

3. *we do $c'n$ total work* per level.

4. **Conclude that total work done by ClosestPair in worst case is at most $c'n \times \log_2 n$. Remember those graphs? Which algorithm is better?**

```
ClosestPair(ptsByX, ptsByY, n)
  if n = 1
     return ∞
  if n = 2
     return distance(ptsByX[0], ptsByX[1])

  mid ← ⌈n/2⌉ − 1
  copy ptsByX[0 ... mid] into new array XL in x order.
  copy ptsByX[mid+1 ... n − 1] into new array XR in x order.

  copy ptsByY into arrays YL and YR in y order, s.t.
      XL and YL refer to same points, as do XR and YR

  distL ← ClosestPair(XL, YL, ⌈n/2⌉)
  distR ← ClosestPair(XR, YR, ⌊n/2⌋)

  midPoint ← ptsByX[mid]
  lrDist ← min(distL, distR)
  Construct array yStrip, in increasing y order, of all
      points p in ptsByY s.t. |p.x − mid.x| < lrDist

  minDist ← lrDist
  for j in 0 ... yStrip.length − 2 do
     k ← j + 1
     while k ≤ yStrip.length − 1 and
              yStrip[k].y − yStrip[j].y < lrDist do
        d ← distance(yStrip[j], yStrip[k])
        minDist ← min(minDist, d)
        k++
  return minDist
```