CSE347 Class 5

Jeremy Buhler

September 13, 2017

1 Back to Divide-and-Conquer

We're going to circle back now and look at another example of divide-and-conquer.

- One place that you will find a lot of D&C algorithms is *computational geometry*.
- In particular, fast algorithms on points in the Euclidean plane are frequently designed using such methods.
- The "magic" behind such algorithms comes down to facts that can be proved with geometric arguments.
- Today, we're going to see an example of this, in an algorithm for the *closest pair problem*.

First, let's set up the problem.

- Suppose we have n points $p_1 \ldots p_n$ in the plane.
- Each point p_i is defined by Cartesian coordinates (x_i, y_i).
- Hence, for any points p_i and p_j, we can compute their *distance*

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

- **Question**: what is the smallest distance beween any two of these points?
- Naively, we can simply compute distances for all pairs p_i, p_j with $i < j$...
- But this method takes $\Theta(n^2)$ distance computations.
- Is it possible for us to do better?

2 A First Attempt to Divide and Conquer

- A natural way to attempt to divide and conquer is to cut the problem in half.
- In this case, we'll divide points into "left" and "right" halves.

- More precisely, given set P of n points, we sort P by x-coordinate into $p_1 \ldots p_n$, then divide this list into
$$Q = p_1 \ldots p_{n/2}$$
and
$$R = p_{n/2+1} \ldots p_n.$$
- We recursively locate the closest pairs of points in the sets Q and R respectively; say that these pairs are at distances d_ℓ and d_r.
- Finally, we check all pairs of points q, r such that $q \in Q$ and $r \in R$. Suppose the closest pair of such points is at distance d_c.
- The globally closest pair is then at distance $\min(d_\ell, d_r, d_c)$.
- Note base cases:
  - If $|P| = 2$, the unique pair is the closest pair.
  - If $|P| = 1$, we should return ∞, so that the "min" operation never uses the result of this subproblem.
- It's easy to prove that this algorithm still considers every possible pair of points, so must be correct.

OK, what have we achieved?

- Given n points, we recur on two subsets of size $n/2$.
- If P is initially sorted by x-coordinate, we can easily divide it into two halves Q and R while keeping each half sorted in time $\Theta(n)$.
- Finally, we might need to do $n/2 \times n/2$, or $\Theta(n^2)$ comparisons to determine the best cross-subproblem distance d_c.
- Conclude that our running time is described by the recurrence
$$T(n) = 2T(n/2) + \Theta(n^2)$$
which has solution $T(n) = \Theta(n^2)$.
- Oh poop. We haven't done any better than the naive algorithm.

To actually improve matters, we need to add some magic to our structure – the magic of geometry!

3 And Now, Some Geometry

- First, let's augment our input a bit.
- We're going to sort our input points P in *two* ways: by x *and* by y.
- Let P_x be the version of P sorted by x, and let P_y be the version sorted by y.

- If we want to divide P into left and right halves, we can (as observed above) create Q_x and R_x from P_x in time $\Theta(n)$ while maintaining sorted order by x.
- Slightly less obviously, we can create Q_y and R_y from P_y in time $\Theta(n)$ while maintaining sorted order by y. **How?**

- Let x_m be the median x-coordinate of P_x, such that all points in Q_x have x-coordinates $\leq x_m$, and all points in R_x have x-coordinates $> x_m$.
- Walk through P_y in order and allocate each point to Q_y or R_y based on whether its x-coordinate is $\leq$ or $> x_m$.
- Now, if we start with inputs P_x, P_y, we can recur on Q_x, Q_y and R_x, R_y, respectively.
- Note that all this splitting work still takes time $\Theta(n)$.

It's not yet clear how keeping the points sorted by *both* x and y helps us. Here comes the geometry...

- The last step of the algorithm is to compute d_c, the distance between the closest pair of points $q \in Q$ and $r \in R$.
- Our first attempt at D&C failed because we have to check *all* such pairs to find the closest, and there can be $\Theta(n^2)$ of them.
- Can we use geometry to rule out some pairs without actually computing their distances?
- Let $\delta = \min(d_\ell, d_r)$.
- **Lemma**: a pair of points (q, r) cannot be closest unless

$$|x_q - x_m| < \delta \text{ and } |x_r - x_m| < \delta.$$

- **Pf**: If q, r is a closest pair from (Q, R), then $d(q, r) < \delta$.
- Now the horizontal distance $|x_q - x_r|$ is surely $\leq d(q, r)$

- Moreover, q and r are on opposite sides of the vertical line through x_m that divides Q from R.

- Hence, the horizontal distances from each of q and r to x_m are at most $|x_q - x_r| \leq d(q, r) < \delta$. QED

How can we augment the algorithm to exploit the lemma?

- Once we compute δ from the two recursive calls, we can filter P to discard all points whose x-coordinate is at least δ away from x_m.
- This may greatly reduce the number of remaining pairs whose distances we need to check.
- Does it reduce the worst-case asymptotic complexity?

- Alas, no – for a bad arrangement of points, there can still be $\Theta(n)$ points inside the "strip" of width 2δ around x_m, hence $\Theta(n^2)$ distance computations.
- We need a further bit of magic still.

4 Even More Geometry

- If two points (q, r) retained by the previous heuristic are the closest pair, they must be close not only in x but also in y.
- In particular, by the same argument as above, $|y_q - y_r| < \delta$.
- One way to exploit this observation is as follows.
- By the previous lemma, filter P_y to create an array S, sorted by y, that retains only those points within δ of the midline at x_m.
- Now, for each point $s \in S$, compute distances to only those points $t \in S$ above s such that $y_t - y_s < \delta$.

- (Note that we can use S here, rather than separately filtering Q_y and R_y, because we won't get a wrong answer by comparing two points on the same side – it is just wasted effort.)
- Again, this should eliminate distance computations for a lot of pairs. But does it change the asymptotic complexity of the cross-subproblem comparisons? **YES!**
- **Lemma**: for each $s \in S$, there are at most 7 other points $t \in S$ such that $y_t - y_s < \delta$.
- **Pf**: Suppose we draw a box of size $\delta \times \delta$, and no two points in this box are at distance $< \delta$.
- We claim that there cannot be more than 4 points in the box.
- Indeed, divide the box into four quarters of size $\delta/2 \times \delta/2$.

- Two points in the same quarter would be at distance $\leq \delta\sqrt{2}/2 < \delta$.
- Hence, there can be at most one point per quarter.

- Now consider the computation after fixing $s \in S$.

- All the points t that we compare to s must lie in one of two boxes of size $\delta \times \delta$: a box whose bottom is at y_s that lies to the *left* of x_m, or a box whose bottom is at y_s that lies to the *right* of x_m.
- Each of these two boxes can contain at most 4 points, so both together contain only 8 points.
- One of these 8 points is s, so there are at most 7 candidates for the point t across both boxes. QED
- **Cor**: If we check only pairs of points in S whose y-coordinates differ by $< \delta$, we compute at most $7n$ distances.
- $7n$ is a heck of a lot better than $\Theta(n^2)$, and we do only $\Theta(n)$ additional work to create S from P_y.

OK, let's put it all together. The following algorithm computes the closest-pair distance; it is straightforward to augment it to return the closet pair itself along with the distance.

```
ClosestPair(P_x, P_y)                    ▷ P_x, P_y sorted by x, y
  if |P| = 1
      return ∞
  else if |P| = 2
      return d(p_1, p_2)
  else
      find median x-coordinate x_m of P_x
      divide P_x into Q_x, R_x around x_m
      divide P_y into Q_y, R_y around x_m
      d_ℓ = ClosestPair(Q_x, Q_y)
      d_r = ClosestPair(R_x, R_y)
      δ = min(d_ℓ, d_r)

      S ← {p ∈ P_y | |x_p − x_m| < δ}
      d* ← δ
      for s ∈ S do
          for t ∈ S | y_t − y_s < δ do
              d* ← min(d*, d(s, t))
      return d*
```

- **Claim**: ClosestPair returns the distance between a closest pair of points in P.
- **Pf**: By induction on $|P|$.
- **Bas**: For $|P| \leq 2$, correct by inspection.
- **Ind**: By the IH, we may assume that d_ℓ, d_r are the closest-pair distances for Q and R, respectively.
- d^* is initially the minimum δ of these distances, and it is updated only if we find a closer pair $q \in Q, r \in R$.
- By Lemma 1, the pair (q, r), if it exists, must lie within the strip S, and by the argument preceding Lemma 2, it must be one of the pairs checked by the nested for loop.
- Conclude that the algorithm correctly returns a distance corresponding to a closest pair. QED

OK, how about our running time?

- We need $\Theta(n \log n)$ time to initially produce P_x, P_y from P.
- Splitting P_x, P_y into its halves takes time $O(n)$.
- Creating S from P_y takes time $O(n)$.
- The nested for-loop takes time $O(n)$ by Lemma 2.

- Hence, the total cost of the algorithm is $O(n \log n) + T(n)$, where

$$T(n) = 2T(n/2) + O(n)$$

  has solution $T(n) = O(n \log n)$.

- Conclude that the whole algorithm takes time $O(n \log n)$.