Closest Pair Algorithm

https://courses.missouristate.edu/anthonyclark/325
Schedule

• Assignment due Monday
• No office hours next Monday or Wednesday
• Activity next Monday will be a practice for the quiz
• Quiz next Wednesday
Outline

Topics and Learning Objectives

• Learn more about Divide and Conquer paradigm
• Learn about the closest-pair problem and its $O(n \lg n)$ algorithm
  • Gain experience analyzing the run time of algorithms
  • Gain experience proving the correctness of algorithms

Assessments

• Closest Pair Activity
Closest Pair Problem

- **Input**: \( P \), a set of \( n \) points that lie in a (two-dimensional) plane

- **Output**: a pair of points \((p, q)\) that are the “closest”
  - Distance is measured using Euclidean distance:

\[
d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}
\]
Closest Pair Problem

- What is the brute force method for this search?
- What is the asymptotic running time of the brute force method?

Can we do better than $O(n^2)$?
One-dimensional closest pair

How would you find the closest two points?
• Sort by position : $O(n \log n)$
  - $p_6$, $p_4$, $p_1$, $p_3$, $p_5$, $p_7$, $p_2$
• Return the closest two using a linear scan : $O(n)$
• Total time : $O(n \log n) + O(n) = O(n \log n)$

Any problems using this approach for the two-dimensional case?
• How do you sort the points?
• *Sorting does not generalize to higher dimensions!*
1. Which two are closest on the y-axis?
1. Which two are closest on the y-axis?
1. Which two are closest on the y-axis?
1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?
1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?

3. Which two are closest?
1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?

3. Which two are closest?
Closet Pair—Two-Dimensions

1. Create a copy of the points (we now have two separate copies of P)
   1. Sort by x-coordinate
   2. Sort other by y-coordinate

\[ O(n \lg n) \]
\[ P : \{ p_0(1,10), \ p_1(2,8), \ p_2(7,3), \ p_3(5,7), \ p_4(8,4), \ p_5(3,5), \ p_6(10,9), \ p_7(9,1) \} \]

Sorted by x coordinate

\[ P_x : \{ p_0(1,10), \ p_1(2,8), \ p_5(3,5), \ p_3(5,7), \ p_2(7,3), \ p_4(8,4), \ p_7(9,1), \ p_6(10,9) \} \]

Sorted by y coordinate

\[ P_y : \{ p_7(9,1), \ p_2(7,3), \ p_4(8,4), \ p_5(3,5), \ p_3(5,7), \ p_1(2,8), \ p_6(10,9), \ p_0(1,10) \} \]
Closet Pair—Two-Dimensions

1. Create a copy of the points (we now have two separate copies of P)
   1. Sort by x-coordinate
   2. Sort other by y-coordinate

   • Can we still end up with a $O(n \lg n)$ algorithm for finding the closest pair?
   • Does the closeness of two points on one axis matter?
1. FUNCTION FindClosestPair(points)
2. points_x = copy_and_sort_by_x(points)
3. points_y = copy_and_sort_by_y(points)
4. RETURN ClosestPair(points_x, points_y)
Closet Pair—Two-Dimensions

1. Create a copy of the points (we now have two separate copies of P)
   1. Sort by x-coordinate
   2. Sort other by y-coordinate

   • Can we still end up with a $O(n \lg n)$ algorithm for finding the closest pair?
   • Does the closeness of two points on one axis matter?

2. Apply the Divide-and-Conquer method
Divide-and-Conquer

1. **DIVIDE** into smaller subproblems
2. **CONQUER** the subproblems via recursive calls
3. **COMBINE** solutions from the subproblems

• How would you divide the problems?
1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?

3. Which two are closest?

4. How would you divide the search space?
1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?

3. Which two are closest?

4. How would you divide the search space?

This is not the average x-value
1. **FUNCTION** ClosestPair(px, py)

2. \( n = \text{px}.\text{length} \) same as \( \text{py}.\text{length} \)

3. **IF** \( n == 2 \)

4. **RETURN** px[0], px[1], \( \text{dist}(\text{px}[0], \text{px}[1]) \)

5.

6.

7.

8. \( \text{pl}, \text{ql}, \text{dl} = \text{ClosestPair}(\text{left}_\text{px}, \text{left}_\text{py}) \)

9.

10.

11.

12. \( \text{pr}, \text{qr}, \text{dr} = \text{ClosestPair}(\text{right}_\text{px}, \text{right}_\text{py}) \)
1. How do we create left\_px?
2. How do we create right\_px?
3. How do we create left\_py?
4. How do we create right\_py?
1. **FUNCTION** `ClosestPair(px, py)`
2. \[ n = \text{px}.\text{length} \]
3. \[ \textbf{IF} \ n = 2 \]
4. \[ \textbf{RETURN} \ \text{px}[0], \text{px}[1], \text{dist}(\text{px}[0], \text{px}[1]) \]
5. \[ \]
6. \[ \text{left}_\text{px} = \text{px}[0 .. < n/2] \]
7. \[ \text{left}_\text{py} = [p \ \text{FOR} \ p \ \text{IN} \ \text{py} \ \text{IF} \ p.x < \text{px}[n/2].x] \]
8. \[ \text{pl, ql, dl} = \text{ClosestPair(} \text{left}_\text{px}, \text{left}_\text{py}) \]
9. \[ \]
10. \[ \text{right}_\text{px} = \text{px}[n/2 .. < n] \]
11. \[ \text{right}_\text{py} = [p \ \text{FOR} \ p \ \text{IN} \ \text{py} \ \text{IF} \ p.x \geq \text{px}[n/2].x] \]
12. \[ \text{pr, qr, dr} = \text{ClosestPair(} \text{right}_\text{px}, \text{right}_\text{py}) \]
Any problems with our current approach?
1. **FUNCTION** ClosestPair(px, py)
2.     n = px.length
3.     IF n == 2
4.         **RETURN** px[0], px[1], dist(px[0], px[1])
5.     **if**
6.     left_px = px[0 ..< n/2]
7.     left_py = [p FOR p IN py IF p.x < px[n/2].x]
8.     pl, ql, dl = ClosestPair(left_px, left_py)
9.    **else**
10.    right_px = px[n/2 ..< n]
11.    right_py = [p FOR p IN py IF p.x ≥ px[n/2].x]
12.    pr, qr, dr = ClosestPair(right_px, right_py)
13.    d = min(dl, dr)
14.    ps, qs, ds = ClosestSplitPair(px, py, d)
15.    **RETURN** Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)

What time complexity does this process need such that the overall algorithm runs in $O(n \lg n)$? **Hint:** think about Merge Sort.
Merge Sort and It’s Recurrence Equation

function MS(array)

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

sort left
sort right
merge (left, right)
return sorted
FUNCTION RecursiveFunction(some_input)
    IF base_case:
        # Usually O(1)
        RETURN base_case_work(some_input)

    # Two recursive calls, each with half the data
    one = RecursiveFunction(some_input.first_half)
    two = RecursiveFunction(some_input.second_half)

    # Combine results from recursive calls (usually O(n))
    one_and_two = Combine(one, two)

    RETURN one_and_two
1. **FUNCTION** ClosestPair(px, py)
2. \[ n = px.length \]
3. **IF** \( n == 2 \)
4. **RETURN** px[0], px[1], dist(px[0], px[1])
5. left_px = px[0 ..< n//2]
6. left_py = [p FOR p IN py IF p.x < px[n//2].x]
7. pl, ql, dl = ClosestPair(left_px, left_py)
8. right_px = px[n//2 ..< n]
9. right_py = [p FOR p IN py IF p.x ≥ px[n//2].x]
10. pr, qr, dr = ClosestPair(right_px, right_py)
11. \( d = \min(dl, dr) \)
12. ps, qs, ds = ClosestSplitPair(px, py, d)
13. **RETURN** Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)

How do we find the closest pair that splits the two sides?
Key Idea

• In ClosestSplitPair we only need to check for pairs that are closer than those found in the recursive calls to ClosestPair.

• This is easier (faster) than trying to find the closest split pair without any extra information!

\[ \delta = \min[d(pl, ql), d(pr, qr)] \]
FUNCTION ClosestSplitPair(px, py, d)
    n = px.length
    x_median = px[n//2].x
    middle_py = [p FOR p IN py IF x_median - d < p.x < x_median + d]

    closest_d = INFINITY, closest_p = closest_q = NONE
    FOR i IN [0..< middle_py.length - 1]
        FOR j IN [1..= min(7, middle_py.length - i - 1)]
            p = middle_py[i], q = middle_py[i + j]
            IF dist(p, q) < closest_d
                closest_d = dist(p, q)
                closest_p = p, closest_q = q

    RETURN closest_p, closest_q, closest_d
Activity
FUNCTION ClosestSplitPair(px, py, d)
    n = px.length
    x_median = px[n//2].x
    middle_py = [p FOR p IN py
                  IF x_median - d < p.x < x_median + d]
    closest_d = INFINITY, closest_p = closest_q = NONE
    FOR i IN [0 ..< middle_py.length - 1]
        FOR j IN [1 ..= min(7, middle_py.length - i)]
            p = middle_py[i], q = middle_py[i + j]
            IF dist(p, q) < closest_d
                closest_d = dist(p, q)
                closest_p = p, closest_q = q
    RETURN closest_p, closest_q, closest_d
Claim

Let \( p \in \text{left}, \ q \in \text{right} \) be a split pair with \( d(p, q) < d \)
Then

A. \( p \) and \( q \) are in \( \text{middle}_\text{py} \), and
B. \( p \) and \( q \) are at most 7 positions apart in \( \text{middle}_\text{py} \)

If the claim is true:

**Corollary 1:** If the closest pair of \( P \) is in a split pair, then our \texttt{ClosestSplitPair} procedure finds it.

**Corollary 2:** \texttt{ClosestPair} is correct and runs in \( O(n \lg n) \)
same recursion tree as merge sort.
Proof—Part A

Let \( p \in \text{left}, \ q \in \text{right} \) be a split pair with \( d(p, q) < d \).

Then

A. \( p \) and \( q \) ∈ \text{middle}_py, and

If \( p = (x_1, y_1) \in \text{left AND} \ q = (x_2, y_2) \in \text{right AND} \ d(p, q) < d \),

Then

\[ x_m - d < x_1 \leq x_m \text{ and } \]
\[ x_m \leq x_2 < x_m + d \]

Otherwise, \( p \) and \( q \) would not be the closest pair with \( d(p, q) < d \).
Proof—Part A

Let \( p \in \text{left}, \ q \in \text{right} \) be a split pair with \( d(p, q) < d \)
Then

A. \( p \) and \( q \in \text{middle}_\text{py}, \) and

If \( p = (x_1, y_1) \in \text{left AND} \ q = (x_2, y_2) \in \text{right AND} \ d(p, q) < d \)
Then

\[
x_m - d < x_1 \leq x_m \quad \text{and} \quad x_m \leq x_2 < x_m + d
\]

Otherwise, \( p \) and \( q \) would not be the closest pair with \( d(p, q) < d \)
Claim

Let \( p \in \text{left}, \ q \in \text{right} \) be a split pair with \( d(p, q) < d \)
Then

A. \( p \) and \( q \) \( \in \) middle\_py, and

B. \( p \) and \( q \) are at most 7 positions apart in middle\_py

If the claim is true:

Corollary 1: If the closest pair of \( P \) is in a split pair, then our ClosestSplitPair procedure finds it.

Corollary 2: ClosestPair is correct and runs in \( O(n \ lg \ n) \) same recursion tree as merge sort
Proof—Part B

How many other points can possibly be in this area?

$p$ and $q$ are at most 7 positions apart in $\text{middle}_\text{py}$

![Diagram showing $p$ and $q$ positions and the area they can occupy.](image)
Proof—Part B

$p$ and $q$ are at most 7 positions apart in $\text{middle}_\text{py}$.

**Lemma 1**: All points of $\text{middle}_\text{py}$ with a $y$-coordinate between those of $p$ and $q$ lie within those 8 boxes.

**Proof**:

1. First, recall that the $y$-coordinate of $p$, $q$ differs by less than $d$.

2. Second, by definition of $\text{middle}_\text{py}$, all have an $x$-coordinate between $\text{xm} += \delta$. 
Proof—Part B

p and q are at most 7 positions apart in middle_py

Lemma 1: All points of middle_py with a y-coordinate between those of p and q lie within those 8 boxes.

Lemma 2: At most one point of P can be in each box.

Proof: By contradiction. Suppose points a and b lie in the same box. Then

1. a and b are either both in L or both in R
2. \(d(a, b) \leq d/2 \sqrt{2} < d\)

This is a contradiction! How did we define d?
Max distance within box is $\frac{d}{\sqrt{2}}$
Claim

Let \( p \in \text{left}, q \in \text{right} \) be a split pair with \( d(p, q) < d \). Then

A. \( p \) and \( q \) ∈ \text{middle}_py, \text{ and}
B. \( p \) and \( q \) are at most 7 positions apart in \text{middle}_py

If the claim is true:

**Corollary 1**: If the closest pair of \( P \) is in a split pair, then our \text{ClosestSplitPair} procedure finds it.

**Corollary 2**: \text{ClosestPair} is correct and runs in \( O(n \lg n) \) same recursion tree as merge sort
Closest Pair

1. Copy \( P \) and sort one copy by \( x \) and the other copy by \( y \) in \( O(n \log n) \)
2. Divide \( P \) into a left and right in \( O(n) \)
3. Conquer by recursively searching left and right
4. Look for the closest pair in middle\_py in \( O(n) \)
   - Must filter by \( x \)
   - And scan through middle\_py by looking at adjacent points
Closest Split Pair
FUNCTION ClosestPair(px, py)

n = px.length

IF n == 2
    RETURN px[0], px[1], dist(px[0], px[1])

left_px = px[0 ..< n//2]
left_py = [p FOR p IN py IF p.x < px[n//2].x]
pl, ql, dl = ClosestPair(left_px, left_py)

right_px = px[n//2 ..< n]
right_py = [p FOR p IN py IF p.x ≥ px[n//2].x]
pr, qr, dr = ClosestPair(right_px, right_py)

d = min(dl, dr)
ps, qs, ds = ClosestSplitPair(px, py, d)

RETURN Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)
FUNCTION MergeSort(array)

n = array.length

IF n == 1
    RETURN array

left_sorted = MergeSort(array[0..<n//2])
right_sorted = MergeSort(array[n//2..<n])

array_sorted = Merge(left_sorted, right_sorted)

RETURN array_sorted

T(n) = 2 T(n/2) + O(n) = O(n \text{ lg } n)
FUNCTION RecursiveFunction(some_input)

IF base_case:
    # Usually O(1)
    RETURN base_case_work(some_input)

# Two recursive calls, each with half the data
one = RecursiveFunction(some_input.first_half)
two = RecursiveFunction(some_input.second_half)

# Combine results from recursive calls (usually O(n))
one_and_two = Combine(one, two)

RETURN one_and_two

T(n) = 2 \cdot T(n/2) + O(n) 
= O(n \log n)