Comparing Algorithms for Finding the Closest Pair of Points in a 2D Plane

David Corbin, Mitchell Harvey and Kyu-Hyeon Lee
COMP 422, Grove City College

November 10, 2019

Introduction

The problem we are trying to solve is given n points in metric space, we want to find a pair of points with the smallest Euclidean distance between the points (including points at the same position).

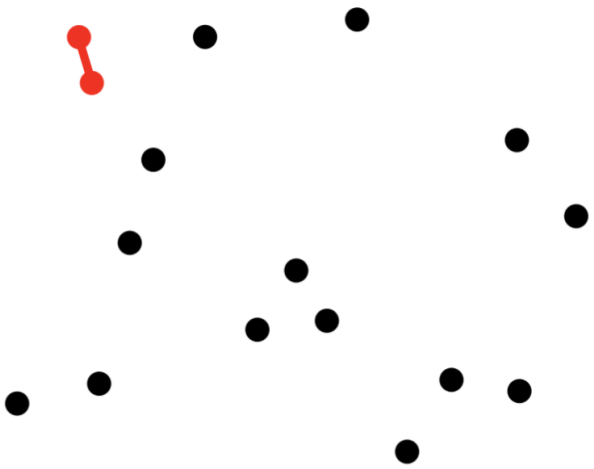

Figure 1. The solution to finding the two closest points in a 2D plane.

The closest pair problem is an interesting algorithmic problem with many practical applications. For one, air traffic control systems use the closest pair of points algorithm and similar algorithms to direct planes on the ground and in the air. The problem itself comes from the field of computational geometry, the branch of computer science devoted to the study of algorithms that can be stated in terms of geometry.

Finding the Distance Between Two Points

To find the distance between two points on a 2D plane, we use the distance formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Geometrically, the distance formula is the length of the hypotenuse of a triangle.

Brute Force Algorithm

The straightforward solution is to check the distance between every possible pair of points and find the shortest distance. This can be done in a brute force way with a nested for loop.

Input: An array of points; each point having an x and y value
Output: The two points with the smallest distance between them

Pseudocode for the Brute Force Algorithm

Listing 1: Pseudocode for the Divide and Conquer Algorithm

```
1 findClosestPoints(points[]):
2         minDistance = Infinity
3         closestA = null
4         closestB = null
5         for a in (0..points.len-1):
6                 for b in (a+1..points.len-1):
7                         d = dist(points[a], points
                              [b])
8                         if (d < minDistance):
9                                 minDistance = d
10                                closestA = points[
                                      a]
11                                closestB = points[
                                      b]
12        return [closestA, closestB]
```

Explanation

The algorithm starts by defining the minimum distance to infinity (on line 2) so that the first two points compared will become the minimum distance. The two closest points, *closestA* and *closestB*, are set to null initially because there are no points checked before the algorithm starts.

The first for loop (on line 5) loops through the array of points starting with the first element in the array. The most naive brute force solution would go through all the points in the array and check all points in the array starting at the beginning of the array each time. In this case, you would check the distance between point A and B and would check the distance between point B and A. Of course, those distances are the same and checking both directions does a significant amount of extra work. This would result in exactly $n^2 - n$ operations or $\theta(n^2 - n)$ runtime. An optimization would be not checking the distance between two points that have already been checked as shown in the pseudocode above.

The inner loop on line 6 loops through all the remaining elements to check the distance between each remaining point in the 2D plane. You can see that we are not checking the distance between points that have already been checked. This slight change results in half the number of checked elements or $(n^2 - n)/2$ operations although this is still an order of n^2 algorithm.

On line 7, we check the distance between the points with the distance formula. This is a constant time operation. The remaining lines of the inner for loop check if the distance is less than the previous minimum distance and update the minimum distance if it is.

Runtime of the Brute Force Algorithm

This results in a time complexity of $\theta(n2)$. This is because we loop through every point, then inside that loop we loop through every other point, and then compare the distance of those two points. The brute force algorithm is in place and the memory consumption $\theta(n)$.

Divide and Conquer Solution

To get a better algorithm, we look to use a divide and conquer approach. We use an algorithm that is recursive, splitting the problem into two equal subproblems.

Summary of the Algorithm

1. Copy the input points into two arrays A_x and A_y.
2. Sort A_x by x coordinates, and A_y by y coordinates.
3. Divide A_x into two equal subsets, p_{left} and p_{right}.
4. Find some vertical line mid_x so that all the points in $p_{left} <= mid_x$ and all the points in $p_{right} > mid_x$.
5. Find the minimum distance between the points in each subset, d_{left} and d_{right}.
6. Find the minimum of d_{left} and d_{right} called d.
7. Get the subset of points from A_y within d distance of mid_x called p_{center}.
8. Check the distance of every point in p_{center} to the next seven points based on decreasing y coordinate. Call the minimum distance d_{center}.
9. Return the minimum of d and d_{center}.

Explanation

We first make two arrays containing the same points. We sort the first array by x coordinates, then sort the second array by y coordinates. The reason we have both arrays is important later.

Note that we can't afford to sort the arrays and subarrays inside the loop and attain our intended time complexity. Thus, we must presort the arrays so that we can achieve $O(n \lg n)$ time complexity.

We then divide the array of points into two equal subarrays and call the

vertical line that separates the halves L. Then we recursively find the pair of closest points in each half, d_{left} and d_{left}.

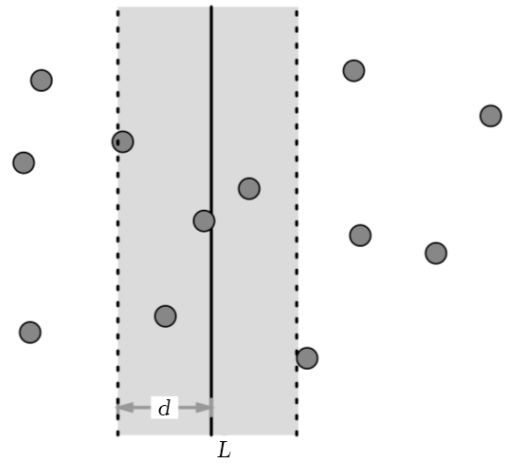

Figure 2. Divide the set of points in the middle and find the closest points in each half.

We then find the minimum distance between the two closest points in each half; $d = min(d_{left}, d_{right})$. However, we're not done yet because there could be points that were split between the two subsets that are closer than the closest points in each subset. Since we know that we already have two points that are d units away from each other, we only need to check for points that are within d units away from L. We call this subset of points p_{center}.

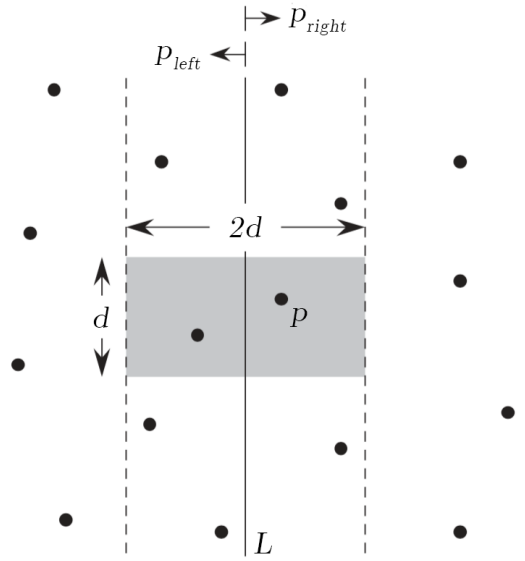

Figure 3. We need only check points within d to the left and right of L.

We copy from A_y to get the points in p_{center}. This way p_{center} is already sorted by y coordinate. Then our subset p_{center} is formed in $\theta(n)$ time, which is important to achieve $O(n \lg n)$ total runtime.

Finding the minimum distance by checking all possible distances between points would not let us complete this step in linear time, so we need a better solution. It turns out we can actually find this minimum distance in linear time:

We already know that the minimum distance we have found so far is d, so we only are interested in distances that are shorter than this. Because of this qualification, we actually only need to check a maximum of seven distances for each point. We can prove this geometrically as shown in Figure 4. If every point in p_{left} and in p_{right} in is exactly d distance apart, then any possible closer points to a point p in p_{center} will be one of the following seven points. This is because the seventh point has to be at least d distance away. Using this to our advantage, we only check the next seven points for each point in p_{center}. This means that this step is completed in $O(n)$ time.

After we complete this step we just have to compare d to the smallest distance

we found in p_{center} and return the smallest distance.

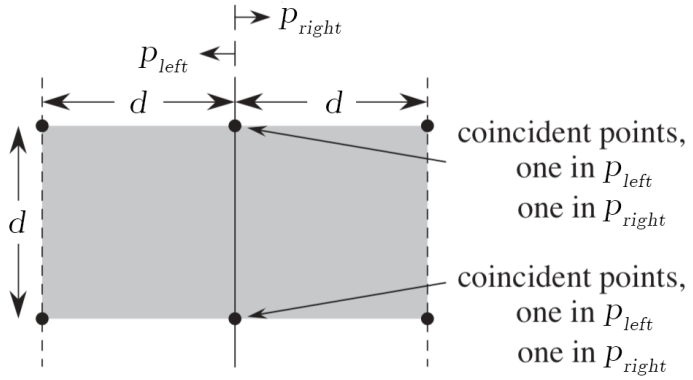

Figure 4. Check a maximum of seven distances for each point.

Runtime of the Divide and Conquer Solution

We can find the total runtime of the algorithm by first finding the runtime of the recursive steps:

1. Divide A_x into two equal subsets, p_{left} and p_{right}.
2. Find some vertical line mid_x so that all the points in $p_{left} \leq mid_x$ and all the points in $p_{right} > mid_x$.
3. $2T(n/2)$: Recursively find the minimum distance between the points in each subset, d_{left} and d_{right}.
4. Find the minimum of d_{left} and d_{right} called d.
5. $\theta(n)$: Get the subset of points from Ay within d distance of mid_x called p_{center}.
6. $O(n)$: Check the distance of every point in p_{center} to the next seven points based on decreasing y coordinate. Call the minimum distance d_{center}.
7. Return the minimum of d and d_{center}.

We can represent this as a recurrence: $T(n) = 2T(n/2) + O(n)$. Then by using the master method we find that this recursive section has a time complexity of $O(n \lg n)$.

Then our full algorithm can be broken down into these steps as follows:

1. Copy the input points into two arrays A_x and A_y.
2. $O(n \lg n)$: Sort A_x by x coordinates.
3. $O(n \lg n)$: Sort A_y by y coordinates.
4. $O(n \lg n)$: Recursively find shortest distance of points.

So the total runtime is $O(n \lg n)$. We cannot give an accurate big theta bound, since when we check the points in the center, there is no guarantee of how many points we will need to check in p_{center}.

Listing 2: Pseudocode for the Divide and Conquer Algorithm

```
findClosestPoints(points[], ypoints[]):
    // If 3 points or less then just return
       shortest distance
    if (points.len <= 3):
        minimum = Infinity
        for a in (0..points.len-1):
            for b in (a+1..points.len-1):
                if (a == b): continue
                minimum = min(minimum, dist(a, b))
        return minimum

    // Split points into two equal subarrays
    mid = floor((points.len-1) / 2)
    midX = (points[mid].x + points[mid + 1].x) / 2
    leftPoints = points[0..mid]
    rightPoints = points[mid+1..points.len-1]
    // Get points on each side sorted by y
       coordinate
    leftYPoints, rightYPoints = getYPoints(ypoints
       , midX)
```

Listing 2 (Cont.): Pseudocode for the Divide and Conquer Algorithm

```
18
19      // Recurse on the subproblems
20      leftMin = findClosestPoints(leftPoints,
            leftYPoints)
21      rightMin = findClosestPoints(rightPoints,
            rightYPoints)
22
23      // Find the min of the left and right minimum
            distances
24      overallMin = min(leftMin, rightMin)
25
26      // get subset of points within overallMin
            distance
27      // of vertical line at midX (Figure 3)
28      centerPoints = getCenterPoints(yPoints, midX,
            overallMin)
29      // Find minimum distance in center points
30      centerMin = findClosestCenter(centerPoints)
31
32      // Find minimum overall
33      return min(overallMin, centerMin)
34
35      getYPoints(ypoints, midX):
36          rightYPoints = []
37          leftYPoints = []
38          for p in ypoints:
39              if p.x <= midX:
40                  leftYPoints.append(p)
41              else:
42                  rightYPoints.append(p)
43          return leftYPoints, rightYPoints
44
45
46      getCenterPoints(points, centerLine, dist):
47          centerPoints = []
```

Listing 2 (Cont.): Pseudocode for the Divide and Conquer Algorithm

```
48          lowerBound = centerLine - dist
49          upperBound = centerLine + dist
50          for i in (0..points.len):
51               p = points[i]
52               if p.x >= lowerBound && p.x <=
                    upperBound:
53                    centerPoints.append(p)
54          return centerPoints
55
56      findClosestCenter(points):
57          minimum = Infinity
58          for i in (0..points.len):
59              // Check the next 7 points if there
                   are >= 7 points
60              numCheck = min(7, points.len - i - 1)
61              for j in (0..numCheck):
62                   d = dist(points[i], points[i + j +
                         1])
63                   minimum = min(minimum, d)
64          return minimum
```

Real World Performance

We implemented the two algorithms in Python and ran them with various input sizes. As you can see, there are several orders of magnitude improvement in the runtimes of the clever algorithm. It's clear that even for relatively small input sizes, the clever algorithm with a runtime of $O(n \log n)$ performs significantly better than the $O(n^2)$ brute force solution.

Real World Results

Input Size	Brute Force		Clever	
	Runtime (s)	Compares	Runtime (s)	Compares
500	0.251	124750	0.071	5096
1000	0.753	499500	0.082	11401
1500	1.638	1124250	0.115	18520
2000	2.990	1999000	0.127	26771
2500	4.534	3123750	0.176	34937
3000	6.202	4498500	0.215	45650
3500	8.823	6123250	0.206	52302
4000	11.452	7998000	0.300	59959
4500	16.524	10122750	0.315	71821
5000	20.771	12497500	0.290	81356

Extensions of the Closest Pair of Points Problem

While our project only covers finding the closest pair of points over a 2D plane, the algorithm can be extended to find the two closest points in a d-dimensional space. It's trivial to see that the naive solution would have a worst case running time of $O(dn^2)$. With the divide and conquer solution, the worst case running time is $O(n * (log(n))^{(d-1)})$ for a normal d-dimensional space. However, there are point sparsity requirements as the number of dimensions increase to avoid edge cases.

Bibliography

[1] Subhash Suri. *Closest Pair Problem*. UC Santa Barbara.
https://sites.cs.ucsb.edu/~suri/cs235/ClosestPair.pdf

[2] Carl Kingsford. *Closest Pair of Points*. Carnegie Mellon University.
https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/closepoints.pdf

[3] Vassos Hadzilacos. *Algorithm to find the closest pair of points*. University of Toronto.
http://www.cs.toronto.edu/~vassos/teaching/c73/handouts/

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press.

[5] Steve Fortune, John Hopcroft. *A Note On Rabin's Nearest-Neighbor Algorithm*. Department of Computer Science at Cornell University.

[6] David M. Mount. *Design and Analysis of Computer Algorithms*. University of Maryland Department of Computer Science.
http://www.cs.umd.edu/class/fall2013/cmsc451/Lects/

[7] *Computational Geometry*. Purdue University.
https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap15d.pdf